



The Professional and Technical UNIX® Association



Conference Proceedings

Winter 1988

**USENIX Technical Conference
Dallas, Texas – February 9-12, 1988**

UNIX® is a Registered Trademark of AT&T

USENIX

CONFERENCE
PROCEEDINGS

Winter
1988

Dallas,
Texas

USENIX Association

Winter Conference

Dallas 1988

PROCEEDINGS

February 9-12, 1988

Dallas, Texas, USA

For additional copies of these proceedings, write:

USENIX Association

P. O. Box 2299

Berkeley, CA 94710 USA

Price \$20.00 plus \$15.00 for overseas airmail

Cray, Cray-1, Cray-XMP, UNICOS, COS, and AIM Job Accounting are trademarks of Cray Research.
ETHERNET is believed to be a trademark of Xerox, Inc.
FREEDOMNET is a trademark of Research Triangle Institute.
HYPERchannel is a trademark of Network Systems Corporation.
INGRES is a trademark of Relational Technology Inc.
MH is believed to be a trademark of Rand Corporation.
MicroVax, Vaxstation II, Vaxstation 2000, Microvax, Microvax, II, VAX/VMS, RSX-11, Ultrix, VAX, DEC, DECnet, DNA, VAX 11/750, VAX 11/780, UNIBUS, MASSBUS, LAT and GFS are trademarks of DIGITAL Equipment Corporation.
NET/ROM is a trademark of Software 2000, Inc.
ONC, NFS, Sun 3/50, and Sun are believed to be trademarks of Sun Microsystems.
ORACLE is a trademark of Oracle Corporation.
PC/AT, RT/PC, ACIS, PC/DOS, and IBM are believed to be trademarks of International Business Machines.
POP and Post Office Protocol might be trademarks – their owner is unknown.
RFS and UNIX are trademarks of AT&T Bell Laboratories.
SYBASE is a trademark of Sybase, Inc.
TBM and Ampex Terabit Memory are trademarks of Ampex Corporation.
UTS is a trademark of Amdahl Corp.
X Window System, X Windows, Hesiod, Kerberos, Zephyr, SMS, and RVD are trademarks of the MIT Project Athena.

(c) copyright 1988 by The USENIX Association

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain
with the author or the author's employer.

ACKNOWLEDGEMENTS

Sponsored by:	USENIX Association P. O. Box 2299 Berkeley, CA 94710	
Program Chairman:	Rob Kolstad	Convex Computer Corp.
Program Committee:	Rick Adams Steve Bunch Larry Clay Dan Geer J. Scott Goldberg Peter Honeyman Dan Klein Evi Nemeth Michael O'Dell Joe Ramey Charlie Sauer Herb Schwetman Jon Tankersley Dave Taylor David Tilbrook	Center for Seismic Studies Gould CSD MCC MIT Project Athena Telesoft Univ. of Michigan CMU Software Engineering Inst. Univ. of Colorado Maxim Technologies, Inc. Texas Instruments IBM Corporation MCC AMOCO Production Co. Hewlett Packard Laboratories CMU Project Andrew
Tutorial Coordinator:	John L. Donnelly	USENIX Association
USENIX Meeting Planner:	Judith F. DesHarnais	
Proceedings Preparation:	Rob Kolstad	Convex Computer Corp.

Table of Contents

Thursday, February 11, 1988

Plenary Session

Thursday (9:00 am – 9:45 am) Conference Coordinator: Judith DesHarnais, USENIX
 Program Chair: Rob Kolstad, CONVEX Computer Corporation

Opening Remarks
Conference organizers and USENIX board

KEYNOTE ADDRESS
Laszlo Belady, MCC

Morning Session: CMU's Andrew Project

Thursday (9:45 am–12:00 pm) Chair: David Tillbrook, CMU

"Make or Take" Decisions in Andrew	1
<i>James H. Morris (Carnegie Mellon University)</i>	
The Andrew Toolkit - An Overview	9
<i>Andrew J. Palay, Wilfred J. Hansen, Michael L. Kazar, Mark Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader & Thom Peters (Carnegie Mellon University)</i>	
An Overview of the Andrew File System	23
<i>John H. Howard (Carnegie Mellon University)</i>	
Synchronization and Caching Issues in the Andrew File System	27
<i>Michael Leon Kazar (Carnegie Mellon University)</i>	
A Multi-media Message System for Andrew	37
<i>Nathaniel Borenstein, Craig Everhart, Jonathan Rosenberg & Adam Stoller (Carnegie Mellon University)</i>	

Afternoon Session A: Advances in Network Protocols

Thursday (1:30 pm – 5:00 pm) Chair: Peter Honeyman, CITI

An RPC/LWP System for Interconnecting Heterogeneous Systems	43
<i>Jan Sanislo & Mark S. Squillante (University of Washington)</i>	
UNIX Systems as Cypress Implets	55
<i>Douglas Comer & Thomas Narten (Purdue University)</i>	
Special Purpose User-Space Network Protocols	63
<i>Michael J. Yamasaki (NASA Ames Research Center)</i>	
Nest: A Network Simulation and Prototyping Tool	71
<i>David F. Bacon (IBM T. J. Watson Research Center), Jed Schwartz & Yechiam Yemini (Columbia University)</i>	

A Fast Transaction Oriented Protocol for Distributed Applications	79
<i>V. S. Sunderam (Emory University)</i>	
A UNIX Implementation of HEMS	89
<i>Craig Partridge (BBN Laboratories Inc.)</i>	
Building an Equities Trading System in a Distributed UNIX Environment	97
<i>Mark Luppi, Mark Seiden, Joseph Collins, Daniel Fisher, Keith Iverson, Charles Marshall, Josef Sachs & David Shaw (Morgan Stanley & Co.)</i>	
Comparing the Efficiency of the Internet Protocols to DECNET	105
<i>R. L. Murphy (Southwest Research Institute)</i>	

Afternoon Session B: Systems Administration

Thursday (1:30 pm – 5:00 pm)

Chair: Jon Tankersley, AMOCO

PMON: Graphical Performance Monitoring Tool	111
<i>Paul Jatkowski (Rich Inc.) & Mike Akre (AT&T)</i>	
System Administration in a Heterogeneous Network	119
<i>Bob Hofkin & W. Terry Hardgrave (Software Productivity Consortium)</i>	
A Faster UNIX Dump Program	125
<i>Jeff Polk & Rob Kolstad (CONVEX Computer Corporation)</i>	
Automatic Unix Backup in a Mass-Storage Environment	131
<i>Edward R. Arnold & Marc E. Nelson (National Center for Atmospheric Research)</i>	
System Administration Daemons	137
<i>Von Jones (Convex Computer Corp.)</i>	
The Postman Always Rings Twice: Electronic Mail in a Highly Distributed Environment	145
<i>Dave Taylor (Hewlett-Packard Laboratories)</i>	
A User Account Registration System for a Large (Heterogeneous) UNIX Network	155
<i>Joseph N. Pato, Elizabeth Martin & Betsy Davis (Apollo Computer Inc.)</i>	
Project Accounting on UNICOS	173
<i>Charles K. Eaton (General Electric Company)</i>	
Using Groups Effectively In Berkeley Unix	171
<i>Scott D. Carson (University of Maryland)</i>	

Friday, February 12, 1988

Morning Session: MIT's Project Athena

Friday (9:00 am – 12:00 pm)

Chair: Dan Geer

Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD	175
<i>G. Winfield Treece (MIT Project Athena)</i>	
The Hesiod Name Server	183
<i>Stephen P. Dyer (MIT Project Athena)</i>	

Kerberos: An Authentication Service for Open Network Systems	191
<i>Jennifer G. Steiner (MIT Project Athena), Clifford Neuman (University of Washington) & Jeffrey I. Schiller (MIT Project Athena)</i>	
The Athena Service Management System	203
<i>Mark A. Rosenstein, Daniel E. Geer, Jr. & Peter J. Levine (MIT Project Athena)</i>	
The Zephyr Notification Service	213
<i>C. Anthony DellaFera (DEC/MIT Project Athena), Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl & William E. Sommerfeld (MIT Project Athena)</i>	
The X Toolkit: More Bricks for Building User-Interfaces or Widgets For Hire	221
<i>Ralph R. Swick (DEC/MIT Project Athena) & Mark S. Ackerman (MIT Project Athena)</i>	
A Simple X11 Client Program -or- How hard can it really be to write "Hello, World"	229
<i>David Rosenthal (Sun Microsystems)</i>	

Afternoon Session A: Kernel Papers

Friday (1:30 pm – 5:00 pm)	Chair: Michael D. O'Dell, Maxim Technologies
The Counterpoint Fast File System	243
<i>Dr. J. Kent Peacock (Counterpoint Computers)</i>	
UNIX I/O In a Multiprocessor System	251
<i>A. J. van de Goor (Delft University of Technology) & A. Moolenaar (Oce Nederland B. V.)</i>	
Beyond Threads: Resource Sharing in UNIX	259
<i>J. M. Barton & J. C. Wagner (Silicon Graphics, Incorporated)</i>	
Watchdogs: Extending the UNIX File System	267
<i>Brian N. Bershad & C. Brian Pinkerton (University of Washington)</i>	
Invoking System Calls from Within the UNIX Kernel	277
<i>Mike Mitchell, Kent Moat, Tom Truscott & Bob Warren (Research Triangle Institute)</i>	
An Experimental Symmetric Multiprocessor Ultrix Kernel	283
<i>Graham Hamilton & Daniel S. Conde (Digital Equipment Corporation)</i>	
A New Exception Handling Mechanism for the UNIX Kernel	291
<i>Joseph R. Eykholt (Amdahl Corporation)</i>	
Translation Lookaside Buffer Synchronization in a Multiprocessor System	297
<i>Michael Y. Thompson, J. M. Barton, T. A. Jermoluk & J. C. Wagner (Silicon Graphics Computer Systems)</i>	

Afternoon Session B: Potpourri and Battle of the Migrating Processes

Friday (1:30 pm – 5:00 pm)	Chair: Rick Adams, Center for Seismic Stud.
Adding Packet Radio to the Ultrix Kernel	303
<i>Clifford Neuman & Wayne Yamamoto (University of Washington)</i>	
Man-Machine Interfaces for software development environments (HandS)	309
<i>Eiji Kuwana (NTT Software Laboratories), Hironobu Nagano (NTT Software Engineering Center) & Yuzou Nakamura (NTT Software Laboratories)</i>	

A Memory Allocator with Garbage Collection for C	323
<i>Michael Caplinger (Bell Communications Research)</i>	
Rescuing Data in UNIX File Systems (What to do after rm *)	331
<i>Jim Joyce & Bob Nystrom (The Gawain Group)</i>	
How To Steal Code -or- Inventing The Wheel Only Once	335
<i>Henry Spencer (University of Toronto)</i>	
The Integration Toolkit and the Unison Real Time Operating System	347
<i>P. Kim Rowe, D. Graham & A. Donenfeld (Multiprocessor Toolsmiths Inc.) & B. Pagurek (Carleton University (Ottawa))</i>	
Process Migration in UNIX Networks	357
<i>K. I. Mandelberg & V. S. Sunderam (Emory University)</i>	
A Process Migration Implementation for a Unix System	365
<i>Rafael Alonso & Kriton Kyrinis (Princeton University)</i>	
Process Cloning: A system for duplicating UNIX processes	373
<i>Chad Hunter (The MITRE Corporation)</i>	

AUTHOR INDEX

- 221: Ackerman, Mark S.
 111: Akre, Mike
 365: Alonso, Rafael
 131: Arnold, Edward R.
 71: Bacon, David F.
 9: Bader, Miles
 259: Barton, J. M.
 297: Barton, J. M.
 267: Bershad, Brian N.
 37: Borenstein, Nathaniel
 323: Caplinger, Michael
 171: Carson, Scott D.
 97: Collins, Joseph
 55: Comer, Douglas
 283: Conde, Daniel S.
 155: Davis, Betsy
 213: DellaFera, C. Anthony
 347: Donenfeld, A.
 183: Dyer, Stephen P.
 173: Eaton, Charles K.
 213: Eichin, Mark W.
 37: Everhart, Craig
 291: Eykholt, Joseph R.
 97: Fisher, Daniel
 213: French, Robert S.
 203: Geer, Jr., Daniel E.
 251: Goor, A. J. van de
 347: Graham, D.
 283: Hamilton, Graham
 9: Hansen, Wilfred J.
 119: Hardgrave, W. Terry
 119: Hofkin, Bob
 23: Howard, John H.
 373: Hunter, Chad
 97: Iverson, Keith
 111: Jatkowski, Paul
 213: Jedlinsky, David C.
 297: Jermoluk, T. A.
 137: Jones, Von
 331: Joyce, Jim
 9: Kazar, Michael L.
 27: Kazar, Michael L.
 213: Kohl, John T.
 125: Kolstad, Rob
 309: Kuwana, Eiji
 365: Kyrimis, Kriton
 203: Levine, Peter J.
 97: Luppi, Mark
 357: Mandelberg, K. I.
 97: Marshall, Charles
 155: Martin, Elizabeth
 277: Mitchell, Mike
 277: Moat, Kent
 251: Moolenaar, A.
 1: Morris, James H.
 105: Murphy, R. L.
 309: Nagano, Hironobu
 309: Nakamura, Yuzou
 55: Narten, Thomas
 131: Nelson, Marc E.
 9: Neuendorffer, Thomas P.
 191: Neuman, Clifford
 303: Neuman, Clifford
 331: Nystrom, Bob
 347: Pagurek, B.
 9: Palay, Andrew J.
 89: Partridge, Craig
 155: Pato, Joseph N.
 243: Peacock, Dr. J. Kent
 9: Peters, Thom
 267: Pinkerton, C. Brian
 125: Polk, Jeff
 37: Rosenberg, Jonathan
 203: Rosenstein, Mark A.
 229: Rosenthal, David
 347: Rowe, P. Kim
 97: Sachs, Josef
 43: Sanislo, Jan
 191: Schiller, Jeffrey I.
 71: Schwartz, Jed
 97: Seiden, Mark
 97: Shaw, David
 9: Sherman, Mark
 213: Sommerfeld, William E.
 335: Spencer, Henry
 43: Squillante, Mark S.
 191: Steiner, Jennifer G.
 9: Stern, Zalman
 37: Stoller, Adam
 357: Sunderam, V. S.
 79: Sunderam, V. S.
 221: Swick, Ralph R.
 145: Taylor, Dave
 297: Thompson, Michael Y.
 175: Treese, G. Winfield
 277: Truscott, Tom
 9: Wadlow, Maria G.
 259: Wagner, J. C.
 297: Wagner, J. C.
 277: Warren, Bob
 303: Yamamoto, Wayne
 63: Yamasaki, Michael J.
 71: Yemini, Yechiam

THE HISTORY OF THE

REIGN OF

CHARLES THE FIRST

BY

JOHN BURNET

OF

GLASGOW

IN

SCOTLAND

AND

ENGLAND

BY

JOHN BURNET

OF

GLASGOW

IN

SCOTLAND

AND

ENGLAND

BY

JOHN BURNET

OF

GLASGOW

IN

SCOTLAND

AND

ENGLAND

BY

JOHN BURNET

OF

GLASGOW

IN

SCOTLAND

AND

ENGLAND

BY

JOHN BURNET

OF

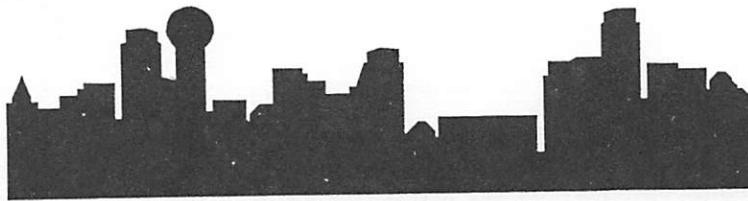
GLASGOW

IN

SCOTLAND

AND

ENGLAND



“Make or Take” Decisions in Andrew

James H. Morris
Information Technology Center
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

ABSTRACT

In creating a software system on top of a rich system like Berkeley UNIX, one has many choices of where to start building a particular facility. In creating the Andrew system we generally chose to maximize our use of existing things in the beginning and gradually replaced components as our understanding increased. In this paper we analyze several examples of this process. Some of the areas discussed are: the programming environment, the file system, protocols, window systems, document editors, the shell, printing, and mail.

Introduction

In 1982 Carnegie Mellon started the Information Technology Center, an IBM-funded development project aimed at creating a prototype university computing environment that came to be called Andrew [1,2]. I was its first director. The charter was to recruit a team of about 35 people, some from IBM, and build the system over a five year period. There were several expectations of what would be done; but we had a great deal of latitude for both the goals and the methods to be employed.

For many years I have been an admirer of F. J. Corbato who supervised the implementation of CTSS and Multics[3] and Butler Lampson who lead many systems projects, notably the Xerox Alto, System [4]. Besides implementing good and useful systems they also wrote some very candid summaries of why they had made certain decisions and how they had fared [5, 6]. If more of us who implemented systems devoted our prose to telling true anecdotes the state of the art would advance faster. In any case, they made a number of observations about implementing software systems with small creative teams which I attempted to apply:

1. Don't count on a half-existent tool for your project. In Corbato's case, this consisted of committing to PL/I as a tool for Multics. Lampson codified such mistakes as "Error 33." For good technical reasons we could have decided to use Modula II as a programming language or Accent [7] as an operating system, but we resisted. I had also seen the programming languages and operating systems intended to support a product actually suck all the energy out of it.
2. While a conservative system design might be the best thing for the users, a university-based project has other constituencies to satisfy: funders, administrators, the general research community,

and the development team itself. All of these groups want the project to be innovative. Corbato used this excuse to explain the elaborateness of the Multics design when it might have been more sensible initially simply to port CTSS to better hardware. Our sin of technical ambition consisted of choosing high-function workstations as the student machine rather than IBM PCs.

3. Try to use one programming environment for all aspects of the project, even if the technical requirements suggest using different ones. If one has a limited amount of talent (who doesn't), one would like to have maximum flexibility in moving people around on a project.
4. Have the programming team use the system they are developing; it helps ensure that it will be usable.

These considerations suggested using UNIX as the operating system for the student computer and the programming environment for the project. Added to these were the compelling arguments about portability and openness. Finally, there was the rather daunting managerial task of forming a team and showing some results quickly; both external and internal pressures called for getting a system deployed at CMU within a few years. If we started designing something from scratch or depended on some less well-known technology we might have spent a few years just getting things sorted out. We felt it was better to get a system out earlier and improve it as we learned more about the real requirements of the users. So off we went into the exciting world of UNIX workstations.

Partly because of this choice, the creation of Andrew has been an exercise in software evolution. The recent IFIP congress had two interesting papers that endorse such an approach. Barry Boehm, the father of software "creationism", better known as the water-fall model, suggested that building prototypes

and re-using software might be a good idea [8]. He codified this idea as the "spiral model." Fred Brooks expressed a more radically evolutionary viewpoint: "The building metaphor has outlived its usefulness...[Software] is grown, not built ... Each added function and new provision for more complex data or circumstance grows organically out of what is already there." [9]

Only an open, flexible system like UNIX permits this sort of approach. The fact that all the software is available in source form is crucial. A vital part of the software development process is *learning*; the development team needs to learn how the existing things work before they can build new, better ones. It is not sufficient simply to use a particular facility, one needs to be able to study it, modify it, and replace pieces of it. We went through this process with many pieces of UNIX; and this paper is an analysis of how it worked.

The Programming Environment

We began with the basic UNIX programming facilities: the C compiler, *dbx*(1), EMACS, *make*(1), and RCS. We have always attempted to get facilities from elsewhere and adapt them when necessary rather than devote design effort to new ones. Shell scripts were written to make RCS work in our distributed environment; but we were able to throw them away later when our central file system began to work. We created a variant of Make that supported simultaneous construction and installation of modules for different machine architectures. We replaced DBX with the Gnu debugger, GDB [10]. Eventually, programmers started using our new editor in preference to EMACS. We have recently decided to adopt the Pmak source control system [11], but have taken it as is, without extensive modifications. It is plausible that spending some more time on our tools might have made us more productive in the long run, but there is also a chance we would still be arguing over how to design the tools.

The File System

While we successfully avoided the impulse to tinker with the basic operating system concepts of UNIX, the file system part seemed to call for some work. Our decision to support a central service, called VICE, for a campus of several thousand workstations demanded certain changes. Generally, however, we were very conservative about changing things that might make existing programs fail.

The Interface

It seemed obvious that the basic concept behind the UNIX file system – a hierarchy of directories and files with occasional links – was a robust and useful one. We have found it very useful in scaling up to a system supporting many users and departments, distributed over many file servers. However, it should be noted that this concept is not immediately obvious to novice users and often causes confusion.

As detailed elsewhere [12], the basic tactic we employed was to perform a minimal amount of surgery on the kernel so that open and close calls re-emerged in a user process that fetched the file from a server to the local disk cache and stored it back to the server. We didn't have the expertise or market ambitions of SUN or LOCUS who substantially reworked the kernel. On the other hand, our plans were more technically aggressive than UNIX United [13] which took a minimalist approach: linking the file systems of relatively autonomous systems.

Obedying the official semantics of the file system doesn't prevent problems, however. Unavoidably, we changed the practical effect of some kernel calls. For example, many UNIX applications fail to check the error returns from *close*, and under VICE there are many times when such an error return occurs. Generally, even though the semantics are the same one cannot employ techniques based on the assumption of fast disks are available when files are shipped over the network.

Besides supporting all the existing UNIX calls we replaced global and world file protection by a scheme based upon access lists. This was deemed important for a large campus system. This addition has been helpful, but sometimes confuses users because it works independently of the normal UNIX protection; little attempt was made to preserve UNIX semantics.

File Servers

At an early stage we decided that file servers were to be special machines, under administrative control of an operations staff. This was done as an overall simplification for the sake of reliability and security. Therefore, there was no requirement that they support general timesharing access. Thus, in principle they needn't run a UNIX operating system at all; they only need to respond appropriately to VICE requests. However, following principle number 3 we choose to adhere to UNIX.

The first version of the file server leaned very heavily on UNIX facilities. We ran a standard UNIX on each file server. Each workstation had a representative process running on any file server it dealt with. This simple structure saved our creating our own process mechanism, and allowed us to lean on some of the other process management facilities of the system. In a similar spirit, we represented the virtual file sub-tree that a file server held by an isomorphic UNIX file tree on that machine's disk. There was actually a second isomorphic set of files containing other information about files and directories necessary for VICE but not supported by UNIX, e.g., access list information and directives pointing to other file servers.

By the time the first version of VICE was operative, it was obvious that it could be greatly sped up. Several things were done to good effect as detailed in [14], including a reduction in our straight-forward use of the UNIX file system. The one-process-per-workstation scheme was dropped; this reduced process

switch overhead and allowed file locking to be performed in a common process. A completely new directory scheme involving mountable volumes was implemented [15]. This allowed us to tune the directory lookup algorithms and avoid the double structure mentioned above. It also required us to implement our own processes for reconstructing the directory structures after a mishap.

In sum, this two stage implementation process worked out very satisfactorily. Trying to do it all at once would have been too hard, especially considering that the team was not particularly expert in UNIX in the beginning. We doubt that anyone could have predicted where the performance bottlenecks would be without building a prototype.

A case can be made for a third stage of implementation which divorces itself from UNIX almost entirely. Although the performance of file servers is now satisfactory, their mean time to recovery is terrible. With each server supporting about 1 GByte, the amount of time for a conventional UNIX salvage operation (*fsck*) is nearly 40 minutes. This salvage is almost always necessary, no matter how seemingly benign the crash, because of UNIX's somewhat casual write-behind method of storing files. There are many more dependable methods of running a file system.

Protocols

One of the major advantages of UNIX for us was that it had an implementation of TCP/IP, the most popular protocol at Carnegie Mellon. Our first pass at the system used TCP sockets for transferring files. We built various remote procedure call (RPC) protocols on top of TCP/IP to support the file system and other activities.

After the first implementation of VICE we discarded the TCP layer and implemented two successive versions of RPC directly on IP. This was made necessary by limitations on the number of TCP connections a process could have. Also we had some of our own ideas of how to make file transfers faster and were able to double the speed.

At the beginning of the project I had hoped that the VICE file system would present a simple, easy to implement, interface on the wire, so that nearly any machine could connect to VICE. There are currently many more IBM PC's and Macintoshes available to students than UNIX workstations, so linking in non-UNIX machines somehow is essential. However, depending upon the established communications mechanisms of UNIX, especially sockets, prevented us from accomplishing this directly. Instead we needed to install various intermediate server machines which ran UNIX, linked into VICE, and spoke a special protocol to the non-UNIX machine. This has been accomplished for PC's. We are still working on a similar solution for Macintoshes.

The User Interface

"But, UNIX is not a user-friendly system for general campus use," said the Vice Provost of Computing. "Not to worry," said I, "We're only going to use it as the operating system and will paper over all the nasties." The idea was firm in my mind, if not everyone else's that UNIX and its various tools were, a mere transitional stage on the way to a truly user-friendly system. This plan was even enshrined in the name of our user interface sub-system, VIRTUE, standing for "Virtue is reached through UNIX and EMACS."

The Window System

Given a large bit-map display and UNIX, the first step is fairly obvious: let each application process talk to the user through a different window on the screen. This natural idea has served well because it made things more intuitive for the expert and novice alike. The expert, who already understands processes, immediately knows the purpose and use of a window. On the other hand, the novice has a visual manifestation of the otherwise mysterious thing called a process. For example, the idea that one process might be stalled or broken even while others are operative is easily reflected in the fact that its window is non-reactive or even disappears. Various process-related control features – priorities, kill operations, etc. – can be associated with windows and save the user learning the various shell arcana.

In later versions we employed some processes that controlled multiple windows and found the users became confused, asking questions like "Why can't I read my next message while the send window is sending a reply to the last one?" or "Why does zapping one window make two disappear?" We would avoid this mode if UNIX processes were not so expensive.

The next choice we made was less obvious: make the window system itself a user process. The basic task of virtualizing the display is obviously an operating system function, so one can argue that the proper thing to do is add the new facilities to the kernel as one would any other driver that gives shared access to a device. However, we were reluctant to fiddle with the kernel for all the usual reasons, including not having access to the sources on certain machines we used. Also, the recently released 4.2 socket mechanism offered the chance to coordinate processes closely. The general idea is that an application writes to the screen by sending commands down a socket and receives keyboard and mouse input by reading a socket. Subsequently the X [16], NeWS [17], and GMW [18] window systems pursued the same strategy with some success.

The obvious objection to the scheme is that it limits the speed with which one can paint the screen. However, that problem was overcome by batching commands. All the commands that the application process sends to the display are actually saved up in its process memory until it issues a command that demands a response from the window system. It

turned out that even a 1 MIP machine had sufficient power to produce respectable results using this scheme for text and line drawings. The hardest things to deal with are complete raster images, but even these can be painted fairly quickly on faster machines, especially if one cheats a little by introducing sockets that allow the software equivalent of DMA; i.e., transferring large blocks of memory between processes on the same machine.

A major problem with this approach has been mouse tracking. Since a scheduled process is responsible for moving the cursor on the screen when the user moves the mouse, the cursor occasionally freezes. This destroys the user's illusion that the mouse is an effective pointing device and is very frustrating. Of course, it would be possible to put basic mouse support into the kernel; but we have not done so for the powerful reasons listed above. Also, we have created elaborate conventions for changing the cursor shape based on its region that a self-respecting kernel might not support. Machines without cursor hardware flash the cursor when things are drawn under it. If the window system is de-scheduled when the cursor is out, this can be a problem.

A more general form of this problem is represented by the kinds of things a Macintosh can do easily. For example, MacPaint allows one to snip out an arbitrary shape and mouse it around the screen smoothly. We have yet to support continuous scrolling in which text rolls by until the mouse button is released. Because people assumed rubber-banding would be slow, it was two years before anyone tried it in the Andrew window system. Once someone figured it out, however, several applications did it. Such challenges can often be met with some work by a clever person who can understand UNIX signals and how the window system deals with them, but the number of such people is vanishingly small. Window systems that allow one to download programs make such stunts somewhat easier.

A significant, much heralded advantage of these window systems is that they allow one to execute the application process on a different machine from the one with the user's screen. This is a direct benefit of using the 4.2 socket mechanism. We have used this feature a great deal in Andrew because we have a large network of machines and a common file system. It is very common for people to use two or three machines at once [19].

Another fact we occasionally ponder is that the UNIX process scheduler still thinks its running a timesharing system and might be improved for workstation use. However, we don't have any clear notions of how it should change.

While virtual memory and multi-processing may have intrinsic value for the user, they present difficult challenges to the user interface designer. Those features take control of performance away from the him. He may get his application running perfectly, but then someone will try to use it an overloaded

workstation and he can't stop them. It's a little like a movie director trying to design a film for which every local projectionist can change the speed of the projector capriciously. Even on an unloaded workstations, the raw process-switching time for UNIX processes precludes our achieving certain effects. Choosing to make the window system itself a scheduled process exacerbates the problem.

The Shell

Naturally, we began by making the shell into a window. Creating more shells created more windows. Thus most of the fiddling around one does with multiple shells became more intuitive for novices. We actually overlaid the shell with another process that maintains the command/response history as an editable text document and thereby provides some of the advantages of paper teletypes as well as the cut and paste operations. Of course, these features already existed in the shell facilities embedded in EMACS.

The more interesting question is why we never moved beyond the shell to provide an icon driven interface as found on the Macintosh Finder. We made two attempts at Finder-like systems, but neither displaced the shell.

The first was Don Z which presented UNIX files in the a Macintosh style. Each subdirectory was treated like a Macintosh folder, and each file was represented by an icon based upon its suffix. Double clicking an icon would invoke the proper program upon it; e.g., a text file always carried the suffix ".d", had an icon like a text page, and invoked the text editor. It even went so far as to be proactive: small speech balloons occasionally arose from random icons inviting the user to activate them. It was table-driven: one could specify the icon, the command to be invoked, the speech balloon, etc. for classes of file names based on a template. One problem was that Don Z had to deal with directories containing huge numbers of files and the iconographic representation completely broke down. The normal searching commands of the shell were more appropriate. Due to other priorities, this experimental system was never adequately developed.

The second interface, Bush, took the approach of drawing directory structures like trees without any fancy iconography. It coped better with large directories and has turned out to be a useful tool for certain directory maintenance operations. Bush does not attempt to understand the types of files.

One reason neither of these could completely displace the shell is their lack of a command script facility. In a variety of systems I have observed, point-and-click-based command drivers, no matter how attractive, have not been able to displace conventional shells because they offered no storable language. Obviously, one can create them; but it is not as obvious as simply storing and re-executing the same language as the user types. It requires a little design work to map every icon-tweaking action into a storable, linear command. There is a deeper problem,

however: if the users all get into the habit of using mouse based commands, they are less likely to learn the linear forms quickly [20].

The worst effect of our continuing to use the C-shell is its consummate weirdness. It has been reported extensively elsewhere how novice and not-so-novice users have gotten totally confused and lost significant amounts of work because a mistyped character invoked an obscure feature of the shell. One doesn't have to iconify things to overcome this problem. The command languages of MS-DOS and TOPS20 show that a relatively safe command processor is possible.

Given a finite amount of talent and time the question of what gets built is one of priorities. We generally favored doing things that added functionality to the system. After our initial work on a window system we decided to adopt the X window system and ignore the command executive problem. We chose to devote much more of our effort to the creation of user interface toolkit and the document editor it supports.

Document Editors

Naturally we began the project using EMACS as the basic text editor. It was in universal use at Carnegie Mellon. It does as well as one can do for character based displays like VT100's and H19's. The only compatible extension one can make is to allow positioning with the mouse. It didn't handle multi-font text or things like drawings and rasters at all; and it wasn't obvious how to extend it gracefully, so we began work on a completely new document editor early.

The first version was called EditText and dealt only with multi-font text, justified in various ways. Menu commands and mouse based-selection were used. There were also facilities for sub-dividing documents into panels. The major accomplishment was simply to paint text on the screen quickly. We used the basic underlying text model of EMACS – a document is a long sequence of characters -- but added inline control characters to signify the beginnings and endings of various styles, e.g., bold. EditText, like Bravo [4], is a sort of mock WYSIWYG editor: the document on the screen contains multiple fonts and formatting similar to what one would see on paper but the appearance is not strictly tied to its eventual printed form. It uses a built-in algorithm to display text on the screen, wrapping lines at window boundaries, and placing text on a screen-pixel basis so as to improve its readability on the screen. The document is unpaginated, like a galley proof. Printing is accomplished by generating *troff* and using the UNIX printing software as is. Thus page breaks, hyphenation, and other things are left to troff. It is possible to view the troff output on the screen, but it can't be modified in that form.

In the process of creating EditText, some popular features of EMACS were omitted. Multiple buffers seemed unnecessary since one could use multiple

windows. The programming language, MockLISP, was not implemented. It might have been simple to replicate MockLISP as long as we were dealing only with text, but we were looking forward to more variegated document elements and didn't see how to meaningfully extend the language. However, EditText did support most of the popular EMACS keyboard commands. Most of the common commands could be invoked by menu, but many more could be invoked by keyboard.

The one-process/one-window model actually led to severe confusion when people used two different editor processes to edit the same document. Possibly because of prior experience with EMACS, they naturally expected edits on either window to effect the same document and were rudely surprised when only the changes to one were saved.

EditText was successful as a document producer of the MacWrite *genre*. Many people used it to produce nice looking documents. However, it was not deemed capable enough to displace existing tools. It could not replace EMACS for programmers and other power-typists because it was a little slower and was not finely tuned for program development. The programmers had become quite fond of a MockLISP package, Electric C, that aided with various syntactic details of the C language; because there was no MockLISP it could not be imported. There was never any attempt to replace troff, Scribe, TeX, or anyone else's favorite document compiler. However, because it used troff as a back end and allowed one to pass through troff directives directly, it seemed like a good tool for troff users; they could use the WYSIWYG features of EditText as far as they went and supplement them with raw troff as needed..

Several specialized editors for drawings, equations, and rasters were also created in the first few years – there were obviously none available in UNIX. The drawing and equation editors used *pic* and *eqn* to print things in the same way as troff.

A second editor, EZ, was created based upon our experience [21]. It was based on a new, more general object-oriented toolkit that allows arbitrary nesting of different document elements. Currently, there are document elements for text, drawings, tables, equations, spread sheets, rasters, animations, and C program text. It added the EMACS multiple buffer features and allowed entirely separate windows to be run from the same process. Aside from being a major advance in terms of multi-media documents, it is preferred to EMACS by many C programmers.

Thus far, the original troff back end printing strategy has been retained. This has been a seductive path of least resistance, but has been unfortunate for a number of reasons. First, the limitations on what troff *et al.* can print has limited our aspirations. For example, thick lines and filled regions are not permitted in any of our drawing editors. Second, the reliability of printing has been adversely affected. Added to the expected problem of dealing with many types of

printers we have the difficulty of coping with the foibles of troff as distributed by three different vendors. Also, the programmers generating the printing directives are not entirely familiar with the many features of the document processors so sometimes fall into traps; e.g., a document with a single quote in it recently caused problems. Furthermore, a naive user will occasionally stumble into troff to his regret – often by copying part of a document from someone else who might have known what they were doing. Tracking down printing bugs is a nightmare when there are so many poorly understood steps in the process. Thirdly, depending on UNIX printing software has given us an excuse to avoid the significant problems associated with creating a more faithful WYSIWYG editor – something that many users strongly desire. As long as a black box like troff controls the printed output there is no way to present a faithful interactive rendering of a printed representation.

A General Problem

The managerial problem is crystallized by the following dialog I recently had about our editor:

Q: How do I do a global replace of “UNIX” by “UNIX”?

A: The best I can think of is to save the file and run the following commands on it:

```
sed -e "s/UNIX/\\smaller{UNIX}/g" file.d>foo
mv foo file.d
```

This answer is perfectly reasonable and satisfactory if one is a UNIX person, but it isn't considered a good answer for the freshman History major. Given that our goal is to make an editor that serves the needs of novices, I would have been (strangely) happier with the answer “You can't do it today,” because then someone might someday spontaneously add a facility to the editor to allow it. There are many of examples similar to this one. The very openness and richness of UNIX means that clever people can find many ways to solve a given problem, so the hope of tricking such clever people into providing solutions for less clever ones is vain. “Why don't they just learn UNIX,” has been muttered more than once.

Mail

Mail delivery in a workstation environment is problematical. Sending mail directly to the addressee's personal workstation often fails because it is turned off or in a bad state. Because of this, many communities still seem to use their timesharing systems for reading or sending mail long after other activities move to workstations. The timesharing system is usually up, has an established address, and has familiar software.

We started off using a common machine for mail activities, but as soon as we had a common file system we were able to do mail processing on local machines. We began, like most other UNIX groups, using

sendmail as the routing tool. Every machine ran a copy of it although the controlling file as well as all the spooling directories were kept in VICE. However, we found this solution extremely unsatisfactory. The program was extremely unreliable and consumed significant processing power to do simple things. When anomalies occurred it was hard to reproduce or diagnose them because they occurred on machines remote from the maintainers. Furthermore, since a large percentage of mail sent from Andrew workstations could be expected to stay within Andrew, elaborate and flexible routing methods were not needed so often. Also, we wanted to try some different ways to handle bulletin boards and distribution lists. Finally, the rigid UNIX addressing rules and cumbersome error reporting for mail were irksome. In general, UNIX's mail facilities were considered too primitive for the purposes of Andrew, so there was never any question of retaining them.

Over a period of a few years we made several staged changes to our mail transport system, virtually eliminating the usual UNIX mail facilities. Today it works approximately as follows: [22]

Sending:

1. A user invokes a mail sending operation from a document editor.
2. A preliminary screening of the addressees is done using a data base called the White Pages. If any address is implausible or ambiguous based on local knowledge the user is helped to correct it immediately.
3. The message is stored in his personal out basket directory.
4. A background process on the sender's workstation examines each addressee for the message. If an addressee is an Andrew user, the message is written directly to his mailbox; otherwise, it is put into a queuing directory to be examined by a post-office machine.

Receiving:

A special indicator window tells a user if he has mail waiting in his mail box. When he invokes the *new mail* operation in the mail reading application, the message is moved from the mailbox to his mail database.

A post office machine's major duty is importing and exporting mail from Andrew. It reads the queuing directory mentioned above and uses *sendmail* to direct mail to all the various machines on the networks CMU participates in: DARPA Net, BITNET, and UUCP. It has the mail address *andrew.cmu.edu* for incoming mail and distributes mail to mail boxes, using the White Pages to identify directories. A secondary job for post office machines is to serve as a fall back delivery system for local Andrew mail. If a VICE file server is down, preventing quick delivery, a mail queue on an up server is found, and the post office machine handles it later. If things get really

bad, e.g., the workstation can't reach any file server at all, mail is queued on the sender's workstation and the background process retries delivery later.

Multiple processes have been exploited to good effect by the mail system. The mail preparation and reading programs can be run in foreground while the delivery mechanism runs in background. More significantly, the basic mail data base process can run on UNIX workstations and be accessed by various mail interface programs on PC's and Macintoshes.

Conclusions

The evolutionary, incremental approach to creating the Andrew system has generally worked out very well. We deployed a system to a small but diverse group of about 50 faculty at CMU in January of 1985. It included the initial versions of VICE, the window system, the first document editor, and the sendmail-based version of the mail system. The various enhancements and replacements discussed above have been occurring steadily since then. Today there are nearly 500 workstations and over 6,000 registered users of Andrew at CMU. Growing the system on UNIX has given us great flexibility and leverage.

However, here are some cautionary notes related to the four precepts of the introduction:

1. UNIX, its utilities, and other things we depend on, have some bugs, but our programming team is ill-equipped to fix them, especially on each of the three different machine architectures we employ. If you are responsible for delivering the total system and choose to build an existing thing, you inherit responsibility for it but might not have the expertise to cope.
2. Our commitment to high-function workstations, multi-processing, and virtual memory has placed us outside the main stream of personal computer software development. The market power of 11 million installed MS-DOS systems can make good software of certain types appear faster than can any machine feature or programming tool.
3. Having a single programming environment for all parts of the system obviously limits things. Ideally, we would be supplying a transaction-based file system and a Macintosh-like user interface.
4. The development team using the system they develop works perfectly if they are the only ones to ever use it, but when their needs and skills diverge from the intended users trouble can develop.

Finally, there is the old but useful aphorism, "The best is the enemy of the good." UNIX presented us with many good solutions to users needs. If one has higher aspirations, however, it takes force of will to undertake a completely new implementation.

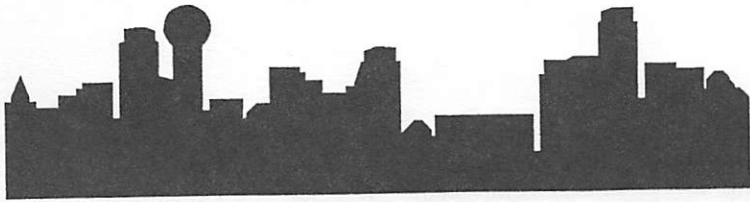
Acknowledgments

Richard Cohn, Fred Hansen, Christine Neuwirth, David Rosenthal, Mahadev Satyanarayanan, and Zalman Stern made some very helpful comments on an early draft of this paper.

References

- [1] Morris, et al., "Andrew: A Distributed Personal Computing Environment", *Communications of the ACM*, March 1986, 29 (1)
- [2] Morris, et al., "Andrew: Carnegie Mellon's Computing System", *Proceedings of the IFIP Congress*, September 1986.
- [3] Corbato, F. J., et al., "Multics - The First Seven Years," *Proceedings of the AFIPS Spring Joint Computer Conference*, 1972, pp 571-583.
- [4] Lampson, B. W. and Taft E., eds., *Alto Users' Handbook*, Xerox Corp., Sept. 1979
- [5] Corbato, F. J. and C. T. Clingen, "A Managerial View of the Multics System Development," in *Research Directions in Software Technology*, P. Wegner, Ed., MIT Press, 1978, pp. 139-158. Also reprinted in *Software Management* (3rd Edition), Donald J. Reifer, ed., IEEE Computer Society, 1986.
- [6] Lampson, B. W., "Hints for Computer System Design", *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, November 1985.
- [7] Rashid, R. and Robertson, G. G., "Accent: a communication oriented network operating system kernel", *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, December 1981.
- [8] Boehm, B. W., "Understanding and Controlling Software Costs", *Proceedings of the IFIP 10th World Congress*, 1986, pp. 703-714
- [9] Brooks, F. P. Jr., "No Silver Bullets - Essence and Accidents of Software Engineering", *Proceedings of the IFIP 10th World Congress*, 1986, pp. 1069-1076.
- [10] Stallman, R. M., *GDB Manual, the GNU Source-Level Debugger*, October, 1986.
- [11] Tilbrook, D. M. and Place P. R. H., "Tools for the Maintenance and Installation of a Large Software Distribution," *Proceedings of the USENIX Technical Conference*, June, 1986
- [12] Satyanarayanan, M. et al., "The ITC Distributed File System: Principles and Design", *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985, pp. 35-50
- [13] Brownbridge, D. R. et al, "The Newcastle Connection", *Software Practice and Experience*, 12:1147-1162, 1982
- [14] Howard, John, et. al, "Scale and Performance in a Distributed File System", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987, pp.1-2
- [15] Sidebotham, R. N., "Volumes: The Andrew File System Data Structuring Primitive", *European Unix User Group Conference Proceedings*, August 1986.

- [16] Scheiffler, R. W., and Gettys, J., "The X Window System," *ACM Transactions on Graphics*, Vol 5, No. 2, April 1986, pp79-109
- [17] Sun Microsystems Inc., "NeWS Manual" 800-1632-10, March 1987
- [18] Hagiya, M., "Introduction to the GMW Window System", Research Institute for Mathematical Sciences, Kyoto University, Preprint RIMS-566, 1987.
- [19] Nichols, David, "Using Idle Workstations in a Shared Computing Environment", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987, pp. 5-12
- [20] Cohn, Richard, Ph.D. Thesis CMU CS Department, in preparation.
- [21] Palay et al., "The Andrew Toolkit: an Overview", *Proceedings of the USENIX Technical Conference*, February, 1988. (this volume)
- [22] Borenstein, et. al, "A Multi-media Message System for Andrew", *Proceedings of the USENIX Technical Conference*, February, 1988. (this volume)



USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

The Andrew Toolkit - An Overview

Andrew J. Palay
Wilfred J. Hansen
Mark Sherman
Maria G. Wadlow
Thomas P. Neuendorffer
Zalman Stern
Miles Bader
Thom Peters
Information Technology Center
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

ABSTRACT

The Andrew Toolkit is an object-oriented system designed to provide a foundation on which a large number of diverse user-interface applications can be developed. With the toolkit the programmer can piece together components such as text, buttons, and scroll bars, to form more complex components. It also allows for the embedding of components inside of other components, such as a table inside of text or a drawing inside of a table. Some of the components included in the toolkit are multi-font text, tables, spreadsheets, drawings, equations, rasters, and simple animations. Using these components we have built a multi-media editor, mail system, and help system. The toolkit is written in C, using a simple preprocessor to provide an object-oriented environment. That environment also provides for the dynamic loading/linking of code. The dynamic loading facility provides a powerful extension mechanism and allows the set of components used by an application to be virtually unlimited. The Andrew Toolkit has been designed to be window system independent. It currently runs on two window systems, including X.11, and can be ported easily to others.

Introduction

During the past five years a number of window systems have been developed for high-resolution bit-mapped graphics displays (Macintosh[1], SunWindows [2], Andrew system[3,4], X Windows[5], NeWS[6]). Each of those systems have included a programmer interface for developing applications. These have been low-level interfaces that have provided a simple graphics abstraction, a method for receiving input events and perhaps some simple components of the system (menus, scroll bars, dialog boxes). In writing to the lower-level interface, application programmers continually replicate the same functional body of code. Further, since the lower-level interface provides few guidelines for the developer, it becomes difficult to build components that can be used by other developers. This again results in replication of large amounts of code that should be reused. It also results in the development of inconsistent applications. Since each window system does provide a full user interface system, application programmers have built functionally equivalent but divergent user interfaces. This leads to chaos for a user community. Because of these

problems, higher-level interfaces have been developed (MacApp[7], SunView[8] the Andrew Base Editor[9], X Toolkit[10]).

The Andrew Toolkit (formerly known as Base Environment 2 or BE2) is a new high-level environment for the development of user interface applications. Built upon the lessons learned over the past four years during the development of a prototype system built at the Information Technology Center (ITC), the toolkit provides a general framework for building and combining components. It is based on a minimal protocol that allows components to communicate with each other about user interface policies, while allowing the developer maximum freedom to determine the actual interactions between components.

The Andrew Toolkit has been built using an object-oriented system, the Andrew Class System, that also provides the ability to dynamically load and link code. This ability provides a powerful extension facility for applications. We have already used this feature to build a generic multi-media editor (EZ) that can edit a wide variety of components by loading

the appropriate code when needed. Further, the dynamic loading facility can be used to add additional components to the basic toolkit without having to rebuild the applications.

The toolkit provides the usual set of simple components (menu, scroll bars, etc) and a number of higher-level editable components including multi-font text, tables/spreadsheets, drawings, equations, rasters and simple animations. The text and table components are multi-media components, in that they allow the embedding other components within their description. The drawing component will soon support this feature.

In addition to the editor, we have developed a number of basic applications including a mail system[11], a help system, a typescript facility that provides an enhanced interface to the C-shell, a *ditroff* previewer, and a system monitor (console) that displays status information such as the time, date, CPU load and file system information. Since both the mail and help applications use the text component for the display of information, they automatically inherit the multi-media functionality of the text component. Examples snapshots of these applications can be found at the end of this article.

We have also developed a number of extension packages. These include a C-language programming component, a compile package, a tags package, a spelling checker, a style editor and a filter mechanism¹.

The original design of the Andrew Toolkit arose from discussions about the development of a text editor that would allow the user to embed other components, such as tables, drawings, rasters, etc. Further we wanted those components to be editable in place. Some of the problems that must be addressed in building such a system include:

- how to resolve the handling of input events between components.
- how to arbitrate the display of menus.
- how to arbitrate the display of the mouse cursor.
- how and when to display components.
- how to determine the size and placement of embedded components.
- how to store the external representation of components.

In examining these problems, we developed a general architecture that allows the inclusion of one component inside another. This allows the developer to build components that can embed arbitrary components without detailed knowledge about the embedded object. Further, new components can be easily included in already existing components without any additional work.

For example, users of the Andrew System can currently compose papers that contain tables,

¹the filter mechanism gives the user the ability to use standard tools on regions of text contained in a file being edited.

equations, drawings, rasters and animations. The text component uses a generic mechanism to include other components. If a new component is developed, it can be included in the text component using that same mechanism. This is an important feature of the system, especially within a university environment. Given our limited resources we knew from the outset that we could not write all the components that were required by the university. For example, members of the music department will want to include musical scores inside of text just as easily as others include tables. Members of the electrical engineering department will want to include circuit diagrams inside of text. The list is essentially limitless.

The dynamic loading feature is essential in extending the toolkit's functionality. If a member of the music department creates a music component and embeds that component into a text component (or any other component that allows embedding of components), the code for the music component will be dynamically loaded into the application. Except for a slight delay to load the code, the user of the editor is unaware that the music component was not statically loaded. The user is also unaware that the music component was not part of the original system. The editor did not have to be recompiled, relinked, or otherwise modified to use the new music component. Further, all users of the text component automatically acquire the ability to use the music component: it can be sent in a mail messages easily as edited in a document.

The Andrew Toolkit has been designed to be window system independent (and to a great extent operating system independent). It currently runs under both the original ITC/Andrew Window System and under X.11. Within the university community, and we believe the more general community as well, there exists a wide range of machines and thus window systems that need to be supported. Within the UNIX operating system community, the X Window System is developing into a de facto standard, although other window systems may eventually rival it. In the lower-end personal computer market there exist the PC and the Macintosh. Even though the price of computers will continue to drop those machines will remain useful environments for many users. Finally there is the development of newer operating systems, such as OS/2, for machines between the low-end personal computers and the high-function UNIX workstations.

Since we could not see the development of a window system standard over that range of machines, we chose to make Andrew Toolkit window system independent. This will allow us potential to support a consistent set of applications over a diverse set of window systems. Clearly the personal computer versions would be more limited, but we consider it to be a great advantage for a user to be able to use a low-cost machine for the normal simple tasks, and then easily move to the more powerful machines when required.

Basic Toolkit Objects: Data Objects and Views

The Andrew Toolkit is based on the development of components that can be used as building blocks for either applications or other more complex components. *Data objects* and *views* are two closely related basic object types within the toolkit. A toolkit component is normally composed of a view/data object pair. The data object contains the information that is to be displayed, while the view contains the information about how the data is to be displayed and how the user is to manipulate the data object (the user interface). For example, the text data object contains the actual characters, style information and pointers to embedded data objects. It also provides ways to alter the data, such as inserting characters and deleting characters. The text view contains information such as the current selected piece of text, the portion of the text that is currently visible, and the location of the text. The text view provides methods for drawing the text, handling various input events (mouse, keyboard, menus), and manipulating the visual representation of the text.

The contents of a data object can be saved in a file, but the contents of the view cannot. The information associated with the view is transient and is valid only during the running of an application. When the application terminates that information has no further meaning. On the other hand, views provide the facility for printing within the Andrew Toolkit.

While it is often the case that a view has an underlying data object, there are many cases when a view will be used to solely provide a user interface function. In such a case there is no underlying data object. The scroll bar is one such example. It only adjusts the information contained in another view.

We have made the view/data object distinction to provide a system where multiple views can simultaneously display the information contained in a single data object. Our design is similar to the Model-View-Controller design used in SmallTalk[12] systems. By comparison, our data objects serve as the models, our views are views, and the controller is distributed between the interaction manager (global decisions) and individual views (decisions between children and parent views).

This separation of concerns has brought us many advantages. For example, in building an editor, we wanted to provide the user with the ability to edit the same information in more than one window. Further, we wanted changes made in one window to be reflected in the other. This case is handled by having two views of the same type, one in each window, displaying information from the same data object. Similarly, we might want to have multiple views of the same type on a single data object in one window. A system like Aldus' PageMaker(TM) could be built under the Andrew Toolkit by allowing the user to specify a set of views and their placement on a page. Some of those views (for example, the text views)

would be examining different sections of the same data object.

It is also possible to have two different types of views displaying information contained in the one data object. Currently the text view is a display-based text processing system. It can be characterized as a semi-WYSIWYG² or a WYSLRN³ view. It displays text with multiple fonts, indentations, etc. but makes no attempt to display the information as it would appear on a piece of paper. This view has been used for the basic text editor as well as the mail and help systems. It has been successful, except in the case the user wants to format the text for printing.

In this case we plan on providing a full WYSIWYG text view. This paper-based text view will be designed to use the same text data object. The user of the system will be able to choose to use either view or perhaps have one window using the normal text view and the other using the WYSIWYG text view. Again changes made in one window will automatically be reflected in the other window.

Just as it is possible to have two different views on the same data object in two windows, it is also possible to have two different views on the same data object within the same window. A text component could have two embedded views on the same data object. For example, the user might want to display a table of numbers and a pie chart representing the table. This could be done by having one table data object and two views, a normal table view and a pie chart view.

Despite its advantages, there is a cost to separating information into data objects and views. Two particular areas of difficulties we have discovered are coordinating data objects and views, and the maintaining stable view state. Each is briefly discussed below.

Our system does not encourage a close connection between the changing of the information contained in a data object and the update of the visual appearance provided by the view. Since only one view will be causing the data object to change, and multiple views may have to reflect the change, a delayed update mechanism must be used. When the user issues a command to a view to alter the underlying data object, the view firsts request that the data object modify itself and then requests the data object to inform all of its views that it has changed. When a view is informed that the underlying data object has changed, it must determine what the change is and update its visual representation appropriately.

The delayed update mechanism is the trickiest challenge in building a data object/view pair. The developer must develop some mechanism with which the view can determine which portion of the data object has changed. This mechanism is normally provided by a set of methods exported by the data object.

²What You See Is What You Get

³What You See Looks Real Neat

It is not considered to be proper behavior for the data object to have detailed knowledge of a specific type of view. This would be one way to handle the delayed update, but would preclude the development of other types of views on the same type of data object.

The second difficulty we faced was retaining the stable (or permanent) state for a view. In the chart example, the underlying data object is a table of values. When a file displaying the chart is saved, only those values (along with the information that a "chart" is viewing the table) is saved. However, the user may have set certain parameters in the chart, such as the way to label the axes. This information is not part of the table data object and would not stored in it. Since a view has no permanent state, information kept in the view, such as axis labelling, would also not be saved. In its simplest form, there is no place to keep this view specific information.

Our solution consists of two parts: additional data objects and the idea of an observer. In the

example above, the chart view would be viewing not a table data object but an auxiliary chart data object. The chart data object would retain information such as axes labelling. In addition, the chart data object would be an observer of the table data object. As information in the table changed, the chart data object would be notified and it, in turn, would notify the chart view. In fact, we do not have specially defined auxiliary data objects. Rather, our update system is based on the observer mechanism, where a data object may be observed by any number of other data objects and views. We have found that this design has permitted a great deal of flexibility and functionality for combining pieces in the toolkit.

Event Processing: The View Tree

Views are used to define the user interface for an application. In defining a user interface, the view must handle both the visual display of information on the screen and the handling of input events that

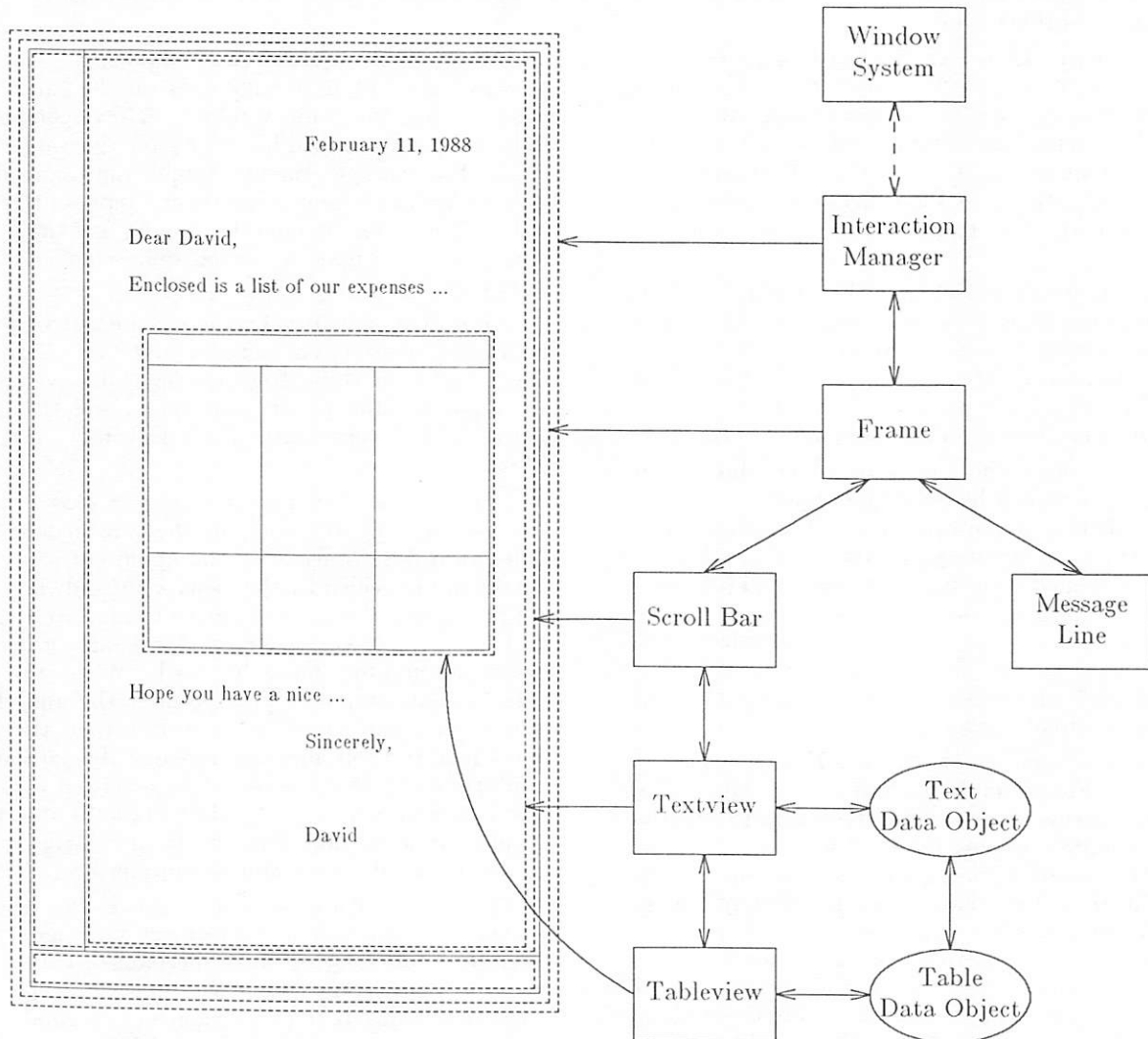


Figure 1

might change that display. Views are organized in a tree structure. Visually, each view is a rectangle and is completely contained in its parent view. At the top of the tree is a view called the *interaction manager* which is a window provided by the underlying window system. The interaction manager has the responsibility of translating input events such as key strokes, mouse events, menu events and exposure events from the window system to the rest of the view tree. The interaction manager is also responsible for synchronizing drawing requests between views. By design, it has one child view, of arbitrary type. That view may have any number of children. Child views are always visually contained within the screen space allocated to their parent, but the toolkit does not define any screen relationship between sibling views.

In general, when an event is received by the interaction manager, the event is passed down to the interaction manager's child. That view determines if it is interested in the event or if it should pass it down to one of its children. This process recurs until some view actually handles the event. By passing the event down the view tree, each parent gets the chance to determine the disposition of the event. The view can use the semantic information associated with itself to make that determination.

Updates to the visual image of an application are handled in a similar fashion. When a view wants to update its image it makes a request to its parent view. That request is usually passed up to the interaction manager which then sends an update event back down the tree. Since a view might be embedded in another view, it does not have complete control over its allocated screen space. The parent might have overlaid some other image on top of the child's image. By posting an update request up the tree and having the update event come back down, the parent can now update its image and the children's images in the appropriate order.

The following figure presents a view tree for a window that contains a scrollable text view that contains a table view. The text view is surrounded by a scroll bar, which is surrounded by a frame. The frame provides a message line view.⁴ The text and table views reference their respective data objects. The lines around the screen image represent the physical area of the image associated with each view.

When a mouse event is received by the interaction manager, it passes the event down to the frame view. The frame determines if it should handle the mouse event directly or if it should be passed down to either of its children. The frame accepts the mouse event directly if it is close to the dividing line between its two children (in this case the user is allowed to adjust the position of the dividing line). If the mouse event is passed down to the scroll bar view, that view will accept the mouse event if it is over the scroll bar

or pass it to its child if it is not. The text view accepts the mouse event if it is not in any of its subviews; otherwise it too passes the event down.

As the mouse event works its way down the view tree, the view determining the disposition of the mouse event only needs to be aware of the location of its children and not the child's type. Similarly, the child needs to have no knowledge about the type of its parent, nor its location in the overall view tree.

The controlling relationship between a view and its children is one area in which the Andrew Toolkit differs from other toolkits. Other systems closely tie the handling of events to the physical relationship of components on the screen. If a component is physically on top of another component it will block the transmission of certain events to the lower component⁵. While this is valid in many circumstances, there are times when it is not. Further, many toolkits use a global analysis of all views in order to process and distribute events. The Andrew Toolkit distributes this authority to each view over its children.

Our toolkit was designed to overcome the limitations that we had experienced with a global, physical model. An early prototype toolkit (the Andrew Base Editor) tied the handling of events to the physical relationship of components on the screen. During the early design phase for the Andrew Toolkit, we attempted to build a drawing editor using that prototype. The drawing editor used the text component to display and edit text within the drawings. The text component was a subordinate of the drawing component. The user of the drawing editor might first enter some text and then place a line over the text. When a mouse event occurs near that line only the drawing component could determine whether the user was selecting the line or the underlying text. This was impossible to accomplish since the toolkit maintained strict, global control over the distribution of input events.

A similar case can be seen in the handling of mouse events by the frame view. The frame physically divides its image into two areas separated by a thin line. In order to allow the user to easily drag that line, the frame allocates a slightly larger area to accept mouse events. That area overlaps the space allocated to the frame's children. If the handling of events was dictated by the screen layout, this interaction would be much more difficult to provide and would require more detailed knowledge of the view tree structure to maintain.

The parental authority is a major architectural concept in the Andrew Toolkit. The discussion above described how this authority is exercised for controlling mouse events. The same mechanism is used between children and parents to negotiate the contents of menus, the display of cursors, the mapping of keyboard symbols and the focus of attention.

⁴The frame (shown in Figure 1), in conjunction with the message line also provides a dialog box facility. To simplify the figure, that detail has been omitted.

⁵X.11 comes very close to handling this correctly except for exposure events which do not propagate to overlapped windows.

The Graphics Layer

The view tree mechanism provides a general mechanism for handling of events in the Andrew Toolkit. It hides from the developer the specifics of the input model used by the underlying window system. In a similar fashion, the toolkit uses a graphics layer to hide the output model of the specific display medium. The display medium is usually the underlying window system, but can also be a printer.

The graphics layer is built using a third type of object, the *drawable*. A drawable contains information about the underlying graphics medium. For a window system, that information normally includes:

- the window to draw in.
- the location of the drawable in that window.
- a small graphics state (e.g. current point, line thickness, current font).
- the coordinate system for the drawable.

The drawable provides a set of drawing operations similar to those provided by the X.11 window system.

Each view contains a pointer to a drawable, which is used for all drawing operations. The developer of a view rarely accesses a drawable directly. All methods exported by the drawable are also provided as part of the view interface.

Separating the view and the drawable will allow us to provide a simple default printing mechanism. When a view receives a print request for a specific type of printer it can temporarily shift its pointer to a drawable for that printer type and do a redraw of its image. We expect to provide this facility in a later version of the toolkit.

External Representation

Most of the problems with embedding components inside other components are solved by the view interface, except for the external representation of the components. As stated earlier, only data object descriptions are written out to files. The toolkit architecture places one requirement on the external representation. When a data object writes out its external representation it is enclosed in a begin/end marker pair. The markers must be properly nested and it must be possible to find all the data associated with an object without actually parsing the data. Those markers provide a tag denoting the type of the object being written and an identification tag that can be used for referencing the data object by other data objects.

Thus the earlier example containing a table embedded in text would have an external representa-

```
\begindata{text, 1}  
text data ...  
\begindata{table, 2}  
the table data goes here ...
```

```
\enddata{table, 2}  
more text data ...  
\view{spread, 2}  
rest of text data ...  
\enddata{text, 1}
```

tion that looks like:

The `\view` construct is specific to the text object and indicates the exact placement within the text of the view (of type `spread`) on the table data object.

The use of nested begin/end markers is the only requirement of for the external representation we also strongly encourage developers to follow the following guidelines:

- use only printable 7-bit ascii characters (including tab, space and new-line).
- keep line lengths below 80 characters.
- make the representation understandable.

The first two suggestions make it possible to transport files across almost all networks (especially as mail). The final suggestion is an attempt to provide an easier recovery mechanism in the rare occurrence when files are partially destroyed. This suggestion only makes sense in the context of the first two suggestions. If the file is being stored using only 7-bit codes and with line lengths than 80 characters then the overhead in making it understandable is small. Of course there are some objects, such as rasters, where this requirement is impossible to meet. However, even in those cases it is possible to make it slightly more comprehensible. For example, the raster format could make sure the bits representing a new row always begin on a new line.

The Object Oriented Environment

The Andrew Toolkit is built using the Andrew Class System (Class). Class provides an object-oriented environment with single-inheritance. The Class language permits the definition of object methods and class procedures. Object methods are similar to C++[13] methods, and a may be overridden in subclasses. Class procedures are similar to SmallTalk's class methods, only they may not be overridden. C procedures for controlling the initialization and disposition of objects are created by the Class preprocessor. Class also provides for the dynamic loading/linking of code.

Class is a C language-based system. It consists of a small run-time library and a simple preprocessor that only preprocesses class header files. The class header files are almost identical to standard C header files except for the inclusion of another type of definition, the class. The preprocessor, generates two include files, an export file (.eh) that is used when defining a class and an import file (.ih) that is used when using a class. The C files written using Class look almost identical to normal C files and are not run through a special preprocessor.

Class is similar in nature to C++. We chose to implement our own system for several reasons:

- We wanted to support dynamic loading/linking. C++ generates code that must be statically linked. Modifying the C++ preprocessor was considered but deemed impractical without getting the changes made in official version. We made some initial inquiries, but could not solve this problem quickly enough for development to continue.
- We wanted to develop a system that could be debugged easily. C++ preprocesses both include and source files. Until a debugger is built that understands the original C++ code, developers would have to understand the transformations made by the C++ preprocessor. This would pose only small problems for the highly experienced developer, but would cause problems for our developer community.
- We wanted as simple a system as possible. We needed an object-oriented environment but not the other constructs built into C++.
- We wanted to be free of any external dependencies that required yet another licensing agreement. We hoped to make the toolkit available to widely available. Requiring another licensed product seemed to be a bad idea.

Even though we did not use C++ to implement the toolkit, the Class system used C++ as a model for its object oriented system. If the above problems were solved (which might now be a possibility) converting to C++ would be a relatively easy (but time-consuming) activity.

Extending the Toolkit

The Andrew Toolkit has been built to be extendible. This is a major feature of the system. The system has been designed in such a fashion that the creator of a data object or view does not have to take any special action to allow that object to be embedded in another object. The data object and view interfaces have been designed to provide the necessary and sufficient set of methods for two objects to communicate without detailed knowledge of each other. Further, those interfaces make it easy for the author of an object to allow it to include other objects. The text object can include any other type of data object. Authors of new objects are strongly encouraged to handle the inclusion of arbitrary objects instead of special casing the inclusion of specific objects.

The dynamic loading/linking feature also provides a low-level extension language for applications built using the toolkit. Sophisticated users can write code (using the class system) to implement new commands. These commands can be bound either to key sequences or to menus. When invoked, the code is loaded and executed.

This feature has also provided us the ability to run all of our applications from a single base program. We have created a program, called *runapp*,

that contains the basic components of the toolkit. The code for each individual application is then dynamically loaded in at run time. Since most UNIX systems do not provide shared libraries, this allows multiple toolkit applications to share a significant portion of code. This leads to performance improvements in a large number of areas:

- paging activity is reduced.
- key portions of the code are almost always paged in thus improving user performance.
- virtual memory use decreases
- file fetch time decreases if running under a distributed file system.
- the file size of an application is reduced.

Window System Independence

The Andrew Toolkit has been designed to be window system independent. To port the toolkit to another window system, six classes must be written, encompassing approximately 70 routines. Of those routines, about 50 routines are normally simple transformations to the graphics layer of the underlying window system. Once those are written, any toolkit application should run in the new environment⁶. The six classes that must be written are:

- *Window System*: this class exists to allow the toolkit to get a handle on the other window system classes listed below.
- *Interaction Manager*: this class provides the interface to the event processing mechanism from the underlying window system. This includes the handling of keystrokes, mouse events and menus.
- *Cursor*: this class provides an interface to defining cursors on the underlying window system.
- *Graphic*: this class provides the output interface to the underlying window system. All drawing operations are made using this class.
- *FontDesc*: this provides an interface to font descriptions.
- *Off Screen Window*: this provides the facility to draw off screen images that can be later included on screen.

Using this facility we are currently able to run applications on two different window systems without any recompilation. Applications are normally configured for one system. However, using the dynamic loading facility, the modules for the other system can be loaded at run time. The choice of window system to use is currently controlled by the setting of an environment variable. With a little more restructuring of the basic code we believe that it will be possible to actually open windows on two different window systems at the same time.

⁶Some applications such as typescript are dependent on the underlying operating system, and will not port quite as easily.

Current Status

The basic toolkit applications (editor, mail, help, preview, typescript, console) have been in general use on the Carnegie Mellon campus for the past four months using the original Andrew window system. The system is actively used by approximately 3000 people. Users are beginning to experiment with the multi-media facility which has only been recently advertised. Within the ITC we are starting to convert to X.11. We hope to be using X.11 within the ITC exclusively by the middle of winter. The timetable for converting the campus to X.11 is currently the summer of 1988. This depends on the conversion of various applications to use the toolkit and the performance of available X.11 systems.

One of the challenges associated with building user-interface software is to make it easy to use for the beginning users while making it powerful enough for experienced users. The prototype editor built at the ITC was highly influenced by the goal of making it easy for the novice user. While we were developing that system and until the release to the ITC of EZ, programmers at the ITC used *emacs* to edit programs. Since the release of EZ, use of *emacs* has dramatically decreased. This has been accomplished without sacrificing the usability of the system by our campus user community.

Conclusion

UNIX, and its software tools approach to computing, provided a new paradigm for building applications. The idea was that portable, general purpose modules could be strung together in different ways to create complex applications without a lot of duplication of effort. The Andrew Toolkit can be seen as an extension of this concept to the graphic workstation environment.

In this way, the Andrew Toolkit is unique among existing toolkits. It provides the usual toolkit functions (text, scroll bars, dialog boxes, etc) but also provides the ability to embed components inside of other components. The architecture for embedding components has been designed to strongly encourage programmers to build new components that can be used in both new and existing applications. The architecture also encourages programmers to develop objects that can include arbitrary components instead of specific ones.

The separation of information into data objects and views provides a highly modularized structure that also supports the building block paradigm. In this way data objects can be used in ways different than originally envisioned by their creator. New views on existing data objects can be created. Existing data objects can also be used as the building blocks for more complex objects. This can be done without using the existing object's companion view.

The Andrew Class System is an essential element in supporting this paradigm. The object oriented

nature of the system allows programmers to easily develop new specialized objects out of existing objects such as the C language component. The dynamic loading facility of Class allows the toolkit to be easily extended by a large community of developers.

Finally, the toolkit is unique among other toolkits in its potential to provide a common base of applications across a diverse set of machines and window systems. We have spent considerable effort to make the system window system independent and believe that it will be important in the future to support the same software base on many systems.

Acknowledgments

The Andrew Toolkit and applications have been designed and developed over the past two and a half years. Many other people have been involved in various phases of the work including Andrew Appel, Nathaniel Borenstein, Mark Chance, Richard Cohn, Curt Galloway, John Howard, Tom Lord, William Lott, Bruce Lucas, David Nichols, Marc Pawliger, and Adam Stoller. The toolkit could not have been built without the earlier work done at the ITC by James Gosling and David Rosenthal. Documentation for the toolkit has been written by Chris Neuwirth and Ayami Orgura.

We are also grateful to our user community, especially our fellow workers at the ITC, who have been forced to be the front-line testing organization for our software over the past four years.

The ITC is a joint project between IBM and CMU. The ITC also receives support from the National Science Foundation. Development and deployment of the toolkit would not have been possible without the support of these organizations.

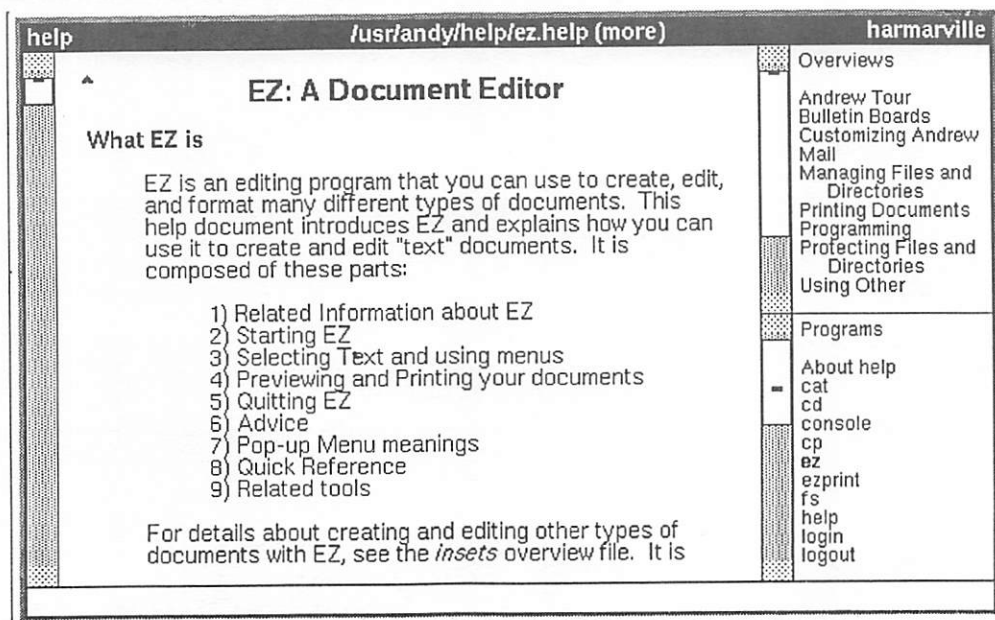
References

1. , *Inside Macintosh*, Addison-Wesley (1985).
2. Anon., *SunWindows System Programmer's Guide*, Sun Microsystems, Inc. ().
3. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Doleson Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM* 29(3) pp. 184-201 (March 1986).
4. James Gosling and David S. H. Rosenthal, "A Network Window Manager," in *Proceedings of the 1984 Uniform Conference*, , Washington, D.C. (January 1984).
5. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics* 5(2) pp. 79-109 (April 1986).
6. , *NeWS Manual*, Sun Microsystems, Inc. (March 1987).
7. K. Shumucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, Hasbrouck Heights, NJ (1986).

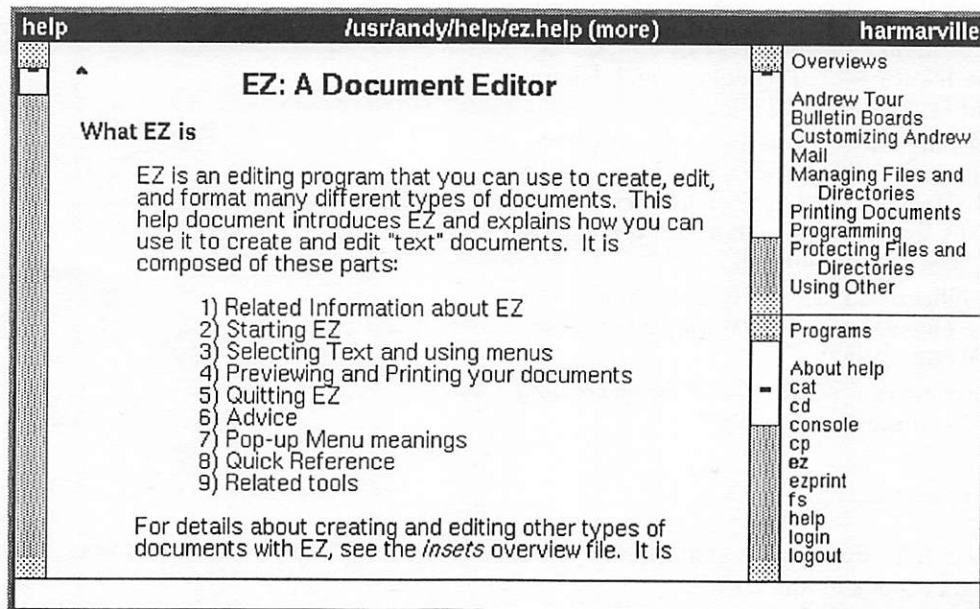
8. Anon., *SunView System Programmer's Guide*, Sun Microsystems, Inc. ().
9. James Goling and David S. H. Rosenthal, "The User Interface Toolkit," in *Proceedings of PRO-TEXT 1 Conference*, (1984).
10. , *X Toolkit Library - C Language Interface*, Massachusetts Institute of Technology and Digital Equipment Corporation (1987).
11. Nathaniel Borenstein, Craig Everhart, Jonathan Rosenberg, and Adam Stoller, "A Multi-media Message System for Andrew," in *Proceedings of the USENIX Technical Conference*, , Dallas, TX (February 1988 (this volume)).
12. Adele Goldberg and David Robson, *SmallTalk-80: The Language and Its Implementation*, Addison-Wesley (1983).
13. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley (1986).

Snapshots

1. Snapshot of a full screen image containing a *console*, a *typescript*, and two *ez* windows. The two *ez* windows show editing a document and a *c* file.



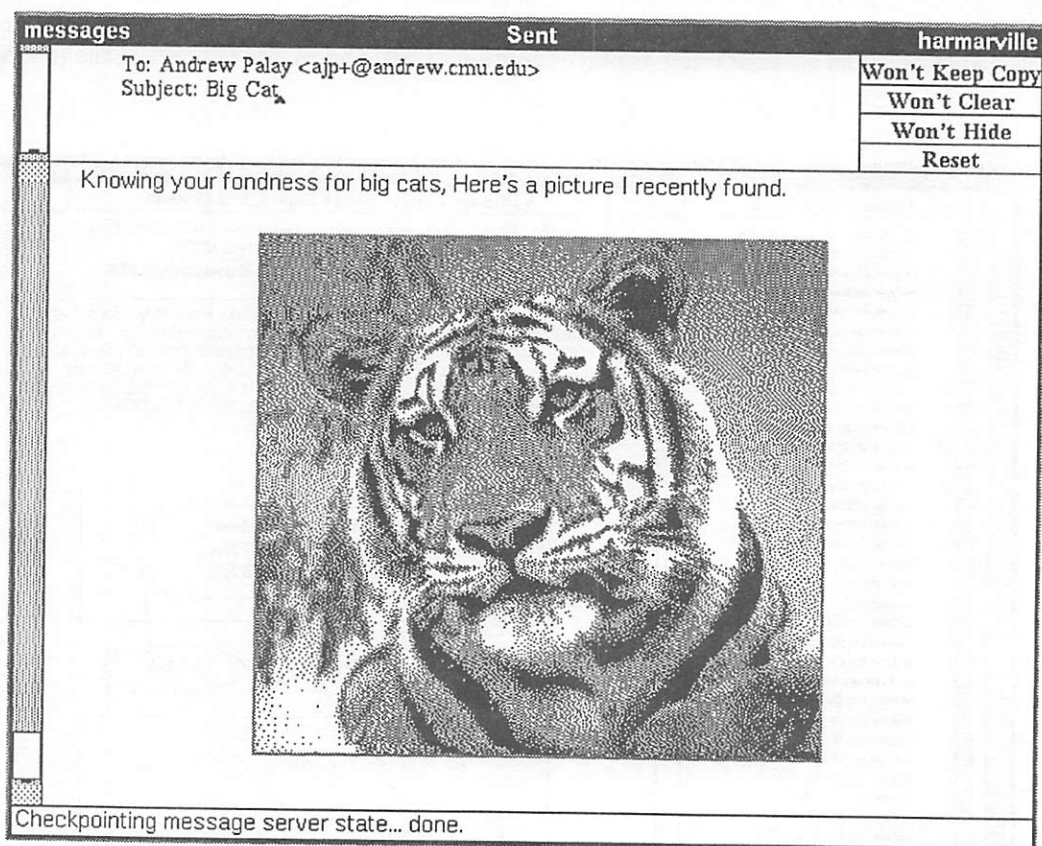
2. Snapshot of a *help* window. Users can see overview information by clicking in the *Overviews* panel on the right. Similarly they can see specific information about a program by clicking in the *Programs* panel.



3. Snapshot of *messages* reading window. The panel on the left gives a list of message folders that can be read. It currently contains a list of all the messages folders available on campus. It can also be set to display the folders a user is subscribed to or just the user's personal folders. The panel at the top left contains the list of messages in the selected folder. The message being displayed contains a drawing within the text of the message.

messages		Version 5.21-N	harmarville
All 1414 Folders		andrew.ms.demo (Local Bboard, 9 of 19 new)	
<ul style="list-style-type: none"> <input type="checkbox"/> andrew.demons.horstable.po2 <input type="checkbox"/> andrew.demons.horstable.po3 <input type="checkbox"/> andrew.demons.horstable.po5 <input type="checkbox"/> andrew.demons.listofflists <input type="checkbox"/> andrew.demons.nntppoll <input type="checkbox"/> andrew.demons.purgesent <input type="checkbox"/> andrew.demons.reindex <input type="checkbox"/> andrew.demons.wp <input checked="" type="checkbox"/> andrew.documentation <input checked="" type="checkbox"/> andrew.expres <input checked="" type="checkbox"/> andrew.gnu-emacs <input checked="" type="checkbox"/> andrew.gripes <input checked="" type="checkbox"/> andrew.helpsys <input checked="" type="checkbox"/> andrew.hints <input checked="" type="checkbox"/> andrew.informix <input checked="" type="checkbox"/> andrew.kermit <input checked="" type="checkbox"/> andrew.kudos <input checked="" type="checkbox"/> andrew.lost <input checked="" type="checkbox"/> andrew.mac <input checked="" type="checkbox"/> andrew.ms <input checked="" type="checkbox"/> andrew.ms.2d <input checked="" type="checkbox"/> andrew.ms.batmail <input checked="" type="checkbox"/> andrew.ms.cui <input checked="" type="checkbox"/> andrew.ms.demo <input checked="" type="checkbox"/> andrew.ms.dow-jones <input checked="" type="checkbox"/> andrew.ms.pcmgs <input checked="" type="checkbox"/> andrew.ms.stats <input checked="" type="checkbox"/> andrew.ms.tech <input checked="" type="checkbox"/> andrew.ms.tech.evs <input checked="" type="checkbox"/> andrew.ms.tech.mac <input checked="" type="checkbox"/> andrew.ms.version-3x <input checked="" type="checkbox"/> andrew.ms.version-4x <input checked="" type="checkbox"/> andrew.musicians <input checked="" type="checkbox"/> andrew.networks <input checked="" type="checkbox"/> andrew.newboards <input checked="" type="checkbox"/> andrew.notes <input checked="" type="checkbox"/> andrew.opinion <input checked="" type="checkbox"/> andrew.opinion.bar-stories <input checked="" type="checkbox"/> andrew.pserver <input checked="" type="checkbox"/> andrew.picture <input checked="" type="checkbox"/> andrew.picture.animals <input checked="" type="checkbox"/> andrew.picture.cartoons <input checked="" type="checkbox"/> andrew.picture.clipart 	<p>✓ 23-Oct-87 <i>What it does, how it works</i> - Nathaniel Borenstein (275)</p> <p>✓ 23-Oct-87 <i>The big picture</i> - Nathaniel Borenstein (2539)</p> <p>✓ 23-Oct-87 <i>The detailed picture</i> - Nathaniel Borenstein (3993)</p> <p>▲ The Andrew message system is, not surprisingly, internally complicated. The drawing below depicts these complications hierarchically. At the top level, it simply shows the five major types of components of the system, which run on five different categories of machines. By using the <i>zip</i> hierarchical drawing editor, you can "zoom in" on the various parts of the picture to see more detail about how each machine's function is structured internally.</p>		

4. Snapshot of *messages composition* window. The message being created contains a raster image.



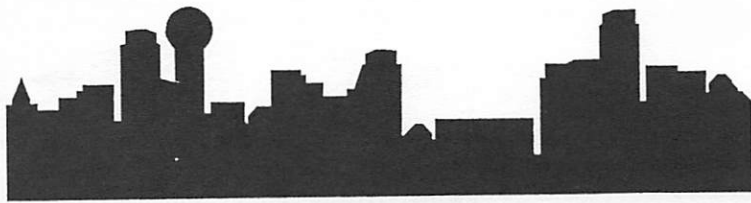
5. Snapshot of an ez window containing a number of embedded objects (text, equations, and an animation) within a table that is contained inside of text.

ez
~/docs/pascal.text
harmarville

This is an example text component that contains a table. The table contains a number of other components including another text component, an equation and an animation. It also shows off the spreadsheet capabilities of the table.

Pascal's Triangle																			
<p>This table contains several descriptions of Pascal's Triangle. It contains a set of equations which defines the values of the triangle. It also contains an animation showing the building of the triangle. Finally there is an implementation of Pascal's Triangle using the spreadsheet facilities of the table object.</p> <p>In order to run the animation, click into the cell and choose the animate item from the menus.</p>	<div style="margin-bottom: 10px;"> $v_{0,j} = v_{i,0} = 0$ $v_{1,1} = 1$ $v_{i,j} = v_{i-1,j} + v_{i,j-1}$ </div> <div style="margin-bottom: 10px;"> </div> <table style="width: 100%; text-align: center;"> <tr><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>2</td><td>3</td><td></td></tr> <tr><td>1</td><td>3</td><td></td><td></td></tr> <tr><td>1</td><td></td><td></td><td></td></tr> </table>			1	1	1	1	1	2	3		1	3			1			
1	1	1	1																
1	2	3																	
1	3																		
1																			
	1	1	1																
	1	2	3																
	1	3	6																
	1	4	10																
	1	5	15																

The End



An Overview of the Andrew File System

John H. Howard
Information Technology Center
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

ABSTRACT

Andrew is a distributed computing environment being developed in a joint project by Carnegie Mellon University and IBM. One of the major components of Andrew is a distributed file system. The goal of the Andrew File System is to support growth up to at least 7000 workstations (one for each student, faculty member, and staff at Carnegie Mellon) while providing users, application programs, and system administrators with the amenities of a shared file system. This overview describes the environment, design, and features of the file system, and ends with a summary of how it "feels" to end users and to system administrators.

History and Goals

The Information Technology Center was started in 1982. It grew out of a task force report on the future of computing at Carnegie Mellon, which recommended that the University move to a distributed computing system based on powerful personal workstations and a networked "integrated computing environment" to tie them together. Overall project goals included advancement of the state of the art in distributed computing, establishing a new model of computer usage, and providing sufficient accessibility and functional richness to become an integral part of users' lives. The integrated computing environment was to allow local control of resources but to permit easy and widespread access to all existing and future computational and informational facilities on campus. Network and file system objectives thought to be necessary to achieve these general goals included high availability, ease of expansion and of incorporation of new technologies, and good performance even during peak periods, with graceful response to overload.

As the project developed, five focal areas emerged: the workstation hardware and operating system, the network, the file system, the user interface, and the message system. This overview deals with the file system, with some passing references to the workstation hardware and the network.

It was generally agreed from the beginning that a high-function workstation should be used rather than a more economical but less functional tool such as a personal computer. The most important feature was a million pixel bit-mapped display; other characteristics (such as a 1 million instruction per second processor and a million bytes of main memory) were selected in proportion to the display's needs. The decision to use UNIX, (specifically 4.2BSD) as an operating system

was made early in the project, based its availability, the desire to use standard tools wherever possible, and the presence in 4.2BSD of desirable features such as virtual memory and networking support.

The network consists of a mixture of Ethernets and Token Rings, tied together by bridges (sometimes referred to as routers). It was built up gradually by inter-connecting pre-existing departmental Ethernets and gradually reorganizing the overall topology into a hierarchy, which at present consists of a campus backbone and a large number of local nets in the individual academic buildings. The local nets are connected to the backbone by fiber optic links which run from the building entry wiring closets to the University Computing Center building. The routers and the backbone are physically located entirely in the UCC building (and one other important academic building), thus greatly simplifying fault isolation and repair.

Given the choice of 4.2BSD, it was easy to select the DARPA TCP/IP protocols as the campus standard. While TCP/IP was not the majority choice of any constituency at Carnegie Mellon, it was at least available for most of the existing computers and operating systems. Other protocols are tolerated on many of the network segments, but the routers operate at the IP packet level.

File System Design

The general goal of widespread accessibility of computational and informational facilities, coupled with the choice of UNIX, led to the decision to provide an integrated, campus-wide file system with functional characteristics as close to that of UNIX as possible. We wanted a student to be able to sit down at any workstation and start using his or her files with as

little bother as possible. Furthermore we did not want to modify existing application programs, which assume a UNIX file system, in any way. Thus, our first design choice was to make the file system *compatible with UNIX at the system call level*. While we realized that there would be some unavoidable differences in performance, in failure modes, and even in some functions, we were determined to minimize them wherever possible. The few differences will be discussed later.

The second design decision was to use *whole files* as the basic unit of data movement and storage, rather than some smaller unit such as physical or logical records. This is undoubtedly the most controversial and interesting aspect of the Andrew File System. It means that before a workstation can use a file, it must copy the entire file to its local disk, and it must write modified files back to the file system in their entirety. This in turn requires using a local disk to hold recently-used files. On the other hand, it provides significant benefits in performance and to some degree in availability. Once a workstation has a copy of a file it can use it independently of the central file system. This dramatically reduces network traffic and file server loads as compared to record-based distributed file systems.[SOSP11] Furthermore, it is possible to cache and reuse files on the local disk, resulting in further reductions in server loads and in additional workstation autonomy.

Two functional issues with the whole file strategy are often raised. The first concerns file sizes: only files small enough to fit in the local disks can be handled. Where this matters in our environment, we have found ways to break up large files into smaller parts which fit. The second has to do with updates. Modified files are returned to the central system only when they are closed, thus rendering record-level updates impossible. This is a fundamental property of the design. However, it is not a serious problem in the university computing environment. The main application for record-level updates is databases. Serious multi-user databases have many other requirements (such as record- or field-granularity authorization, physical disk write ordering controls, and update serialization) which are not satisfied by UNIX file system semantics, even in a non-distributed environment. In our view, the right way to implement databases is not by building them directly on the file system, but rather by providing database servers to which workstations send queries. In practice, doing updates at a file granularity has not been a serious problem.

The third and last key design decision in the Andrew File System was to implement it with *many relatively small servers* rather than a single large machine. This decision was based on the desire to support growth gracefully, to enhance availability (since if any single server fails, the others should continue), and to simplify the development process by using the same hardware and operating system as the workstations. On the other hand, we are interested in

experimenting with a larger server, possibly based on a mainframe, to see if it were more cost-effective or easier to operate. At the present time, an Andrew file server consists of a workstation with three to six 400-megabyte disks attached. We set (and have achieved) a price/performance goal of supporting at least 50 active workstations per file server, so that the centralized costs of the file system would be reasonable. In a large configuration like the one at Carnegie Mellon, we also use a separate "system control machine" to broadcast global information (such as where specific users' files are to be found) to the file servers. In a small configuration the system control machine is combined with a (the) server machine.

Implementation

The Andrew File System is implemented in several parts, which are most conveniently explained by following what happens when an application program attempts to open a file.

The file open system call, issued by the application program, is intercepted by a small "hook" installed in the workstation's kernel. The user's program is suspended and the intercepted request is diverted to a special program, implemented as a user-level process on the workstation. This program, the Andrew Cache Manager, forwards the file request to a file server, receives the file (if everything went well), stores it on the local disk, and sends back an open file descriptor via the kernel to the user's program. The user's program then reads and writes the file locally, without any special help from the kernel, the cache manager, or the file server. Eventually, the close operation is also diverted to the cache manager, which (if the file was modified) sends the updated copy back to the file server.

The Andrew Cache Manager has several other important functions. It keeps copies of recently used files on the workstations' local disk, thus minimizing network traffic for frequently-reused files. It also caches directories and status information and, in fact, performs the entire directory lookup function, again reducing server loads. It maintains much of this status information across file server or network failures and even workstation restarts, simply re-validating cached information the first time it is used. (Once a file has been validated in this way, the server will notify the cache manager if the file changes.) Most of the burden of reproducing UNIX system call semantics falls upon the Andrew Cache Manager as well.

The file server consists of both hardware and software. The software consists primarily of a "file manager" program which runs in a single UNIX process and is internally multiprogrammed by a "light-weight process" package which is capable of handling several requests in parallel.

The operations performed by the Andrew File Manager are reading, creating, or replacing files, reading directories, setting and releasing advisory locks,

and various administrative operations. The manager does the following things to handle a typical operation:

- verify the identity of the user.
- check the user's permission to perform the operation.
- reserve disk space in an update operation.
- perform a bulk transfer of the requested file to or from the workstation as necessary.
- replace the old version of an updated file with the new version atomically.
- send an ending status to the workstation.
- keep a record of the workstation's interest in the file, for use in future update notifications.

Supporting programs include an "authentication manager", an "update manager" and a "status manager". The authentication manager maintains a database of users and authenticates them. The update manager keeps software and internal databases up to date and consistent in configurations with more than one file server. The status manager serves as a centralized collection point for information on the status of the various file servers. This strategy makes it possible for anybody to make occasional or even periodic status queries without overloading the actual file servers.

Features

Here are several specific aspects of the file system worth some more description.

Authentication

When a user logs into a workstation, the workstation goes through an authentication procedure based on the user's password. By authenticating in the file server we avoid depending on the workstation kernel, which is exposed to user modification, for authentication. Although undue reliance on the security of network neighbors is not the only security weakness of distributed systems, we feel it is an important one.

The authentication mechanism is automatic as far as users are concerned. It involves an exchange of encrypted messages between the login program and the authentication manager. A successful exchange establishes the identity of both the user on the workstation and of the authentication manager. It produces some "tokens" which the Andrew Cache Manager uses to establish connections to file servers as it needs them. These tokens can be passed on to other workstations in order to support remote login capabilities. Tokens time out after 25 hours in order to provide some control over the possibility of their being stolen.

Two new utility commands have been provided to make authentication more convenient: *log* re-authenticates a user (for example if the tokens have timed out), and *unlog* discards the tokens, thus rendering the workstation safe for use by other

individuals. It is common to execute an *unlog* command when logging out from a workstation.

Access Lists

We felt that the traditional 12-bit access mode flags of UNIX were inadequate for a campus-wide file system, so we supplemented them by an access list mechanism. An access list specifies various users (or groups of users) and, for each of them, specifies the class of operations they may perform. Access lists are associated with directories. The operations they specify are:

- read any file in the directory
- write (update) any file in the directory
- insert new files in the directory
- delete files from the directory
- lookup files in the directory
- lock files in the directory
- administer the directory (change the access list)

The standard UNIX bits are retained as well, with both conventional and access list permission required to operate on a file. Thus the standard commands such as *chmod* work as specified (although with somewhat dubious security.) The super-user has no special privileges as far as the access list mechanism is concerned. This leads to some difficult issues in the handling of the "setuid" mechanism. Although we have been able to work around the problems we have encountered, this is a significant divergence from standard UNIX semantics.

Access lists are displayed and changed by a user command, *fs*, which passes its requests through the cache manager to the file managers. The *fs* command also provides miscellaneous other operations including listing and controlling the user's disk space quota and other properties of the logical volume (see below), flushing files from the cache, finding out file locations, and general administrative functions.

Logical Volumes

The Andrew File System groups files into aggregates called "logical volumes". A typical logical volume would be a single user's files, or a particular release of the system binary files. Logical volumes are interconnected by a "mount point" mechanism resembling standard UNIX mounts. End users seldom need to be aware of logical volumes. On the other hand, operators and system administrators are almost exclusively concerned with logical volumes rather than with individual user files, as the logical volume is the unit for operations like backup, load and space balancing between file servers, and redundant storage of read-only files.

An important file system operation from an operator's point of view is the creation of a read-only snapshot ("clone"), of any logical volume. This is implemented in such a way that it is quick and inexpensive to make clones; they are used in several important ways. First, new software releases are

typically made by cloning the system binaries. Read-only clones can be replicated on multiple servers, leading to significant increases in availability and performance. (At Carnegie Mellon, there are three servers devoted exclusively to read-only clones of system volumes.) Second, individual users' logical volumes are cloned every midnight. The main reason for this is to get a consistent snapshot which can then be backed up. A secondary benefit is that the clone is made available to each user under the name "Old-Files", so that all the user's previous day's work is available in case of a blunder. This significantly reduces requests to recover backup files.

End Users' Viewpoint

The key fact about the Andrew File System from an end user's viewpoint is that it closely resembles a standard UNIX file system, yet allows the user to sit down at any campus workstation and get at a uniform set of files. In almost all cases it is possible to ignore the distributed implementation, the automatic transfer of files on demand, and the caching mechanism. Application programs imported from more centralized systems can be run without any modification whatsoever and will almost always work. The rest of this section will talk about some differences, but the reader should remember that the differences are exceptions rather than the rule.

The main differences end users see relate to performance and failure modes. There is a noticeable wait when the workstation must fetch a large file. (The effective transfer rate is now about 50 thousand bytes per second, so a million-byte file takes 20 seconds.) Although in absolute terms this is reasonably fast, it is still noticeable and sometimes annoying.

Failure modes in a distributed system are intrinsically different from those in centralized systems. The file system at Carnegie Mellon suffers from occasional failures in the file server hardware, the network connecting workstations to the file servers, and (occasionally) in the power or cooling supply to the entire computing center facility, or the entire campus internet. The design decision to implement the file system with multiple servers is rendered useless when all the servers are disabled by a power or cooling failure, as has happened several times. Much more common, however, are individual server failures. When the cache manager loses contact with a file server it goes through a one-minute timeout, after which it switches to a disconnected mode in which it allows cached files from that server to be read, but not written. If the failed server has your personal files, it won't be long before you get stuck on missing file or a write. In a few cases it is desirable to modify critical application programs, like text editors, to deal gracefully with the resulting error conditions. For example, it is more common for the *close* operation to fail than it is in a conventional UNIX system.

Functionally, the main difference in the Andrew file system is its use of access lists, which we feel have

significant advantage in both flexibility and understandability from a user's standpoint.

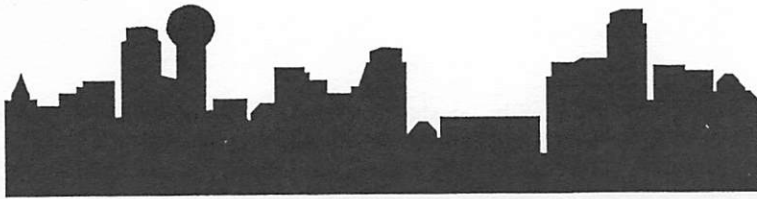
Operators' Viewpoint

Unlike the user, the operator sees the Andrew file system as being very different from a standard UNIX file system. In fact, the main point of contact between them is that UNIX is used as the vehicle for running the file servers. Operators do not (and in fact can not) view or manipulate individual files directly when logged into file servers. (They can, of course, get at individual files through workstations if the access lists allow them to.) Instead, operators and system managers see the file system as a large collection of "logical volumes". Typical operations they perform are to create, adjust, clone, move, and back up logical volumes. Most of these operations are highly automated. For example, there is an operator command *move-vol* which moves a volume from one file server to another. This can be done even when the volume is in use without significant disruption (there may be a period of a few seconds during which the cache manager reports to the user that the volume is busy.)

An interesting question is that of "operating" individual workstations. The Andrew file system is neutral in this respect. There exist at least two strategies, one very centralized and one very distributed. The centralized strategy, which is used for most Andrew workstations today, is heavily dependent on the file system. Individual workstations have a minimum of locally-stored files, such as the UNIX kernel itself or the password list. When a workstation is rebooted a special utility program selectively updates these few files from the central file system. The ordinary user doesn't know the super-user password, and all workstation software is kept up-to-date by system administrators by updating the central directory. At the opposite extreme is the strategy in which the workstation has a full set of local files, maintained entirely by the user, but also uses a connection to the file system for some shared directories. This decreases dependency on the file central system, but fails to take advantage of their benefits such as automatic backup and access to new software releases; it also requires a significantly larger local disk.

A recent development in the Andrew file system is the introduction of a "cellular" mode whereby subsets of the file servers can be administered separately and still be interconnected. The different administrative units may even have separate user registration lists. This feature provides a new dimension of departmental autonomy to the distributed system.

[SUSP11] *Scale and Performance in a Distributed File System*, by J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, in *Eleventh Symposium on Operating Systems Principles*, Austin, Texas, November 1987. Future issue of *ACM Transactions on Computer Systems*.



Michael Leon Kazar
Information Technology Center
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

Synchronization and Caching Issues in the Andrew File System

ABSTRACT

Many distributed file systems go to great extremes to provide exactly the same consistency semantics in a distributed environment as they provide in the single machine case, often at great cost to performance. Other distributed file systems go to the other extreme, and provide good performance, but with extremely weak consistency guarantees. However, a good compromise can be achieved between these two views of distributed file system design. We have built one such system, the Andrew file system. It attempts to provide the best of both worlds, providing useful file system consistency guarantees along with good performance.

Introduction

This paper discusses two important parts of distributed file system architecture: consistency guarantees and performance. When we use the term **consistency guarantees** in the context of a distributed file system, we refer to that part of the file system's specification that describes the state of the file system as seen from one machine in the environment, given the state of the file system at other machines. As an example of a consistency guarantee, Sun Microsystems's NFS file system [Kleiman 86] guarantees that data written to a file by one machine can be read by any machine in the network environment as long as at least 3 seconds have passed after the write call has completed.

The performance of a distributed file system is harder to describe. In this paper, distributed file system performance is characterized by two quantities: the network traffic generated, and the processor load placed on the "bottleneck" machines, usually the file servers. We came to this informal definition of file system performance based on our concern with scaling our distributed file system. We are building a distributed file system intended to scale to thousands of concurrent users; performance bottlenecks in either the processors or the network itself would prevent smooth scaling.

Of course, there are other criteria for measuring file system performance; for example, one might be interested in the smallest possible latency between the arrival of a request and its handling. The statements made here do not directly apply to other definitions of performance. Nevertheless, many of our performance improvement techniques apply to several performance models.

The consistency guarantees provided by a

distributed file system strongly affect the possible implementations for the file system. If one wants to build a high-performance distributed file system, one must begin with consistency semantics that admit a good implementation. Of course, file system consistency semantics also affects the usefulness of a distributed file system for building applications, especially for distributed applications. If a distributed file system provides consistency guarantees that are too weak to use in building a distributed application, then that application will not be able to use the distributed file system for storage and sharing, and the system will be a failure.

This paper discusses the relationship between a distributed file system's consistency semantics and its performance. It begins in Section 2 with a survey of several extant distributed file systems, with particular emphasis on both performance bottlenecks and file system consistency semantics.

The next few sections describe the Andrew file system, a file system that makes different consistency/performance trade-offs from those described previously. Section 3 describes our goals in building the Andrew file system, along with a high-level description of the facilities it provides. Section 4 describes the implementation of the file system, and Section 5 describes our consistency guarantees, and how we implement them efficiently in practice.

Finally, this paper concludes by describing our experiences actually using the Andrew file system, with particular regard to how well the goals enumerated in Section 3 were achieved in practice. The paper concludes that efficiency and file system semantics can be balanced, leading to both good system performance along with a powerful system that can be used for sharing data among many machines

easily.

Other Distributed File Systems

With the increasing use of local-area networks for connecting workstations and mainframe computers together, distributed file systems have been sprouting like dandelions. This section describes several such file systems: Suns' NFS distributed file system, the Apollo Domain system, AT&T's Remote File Sharing file system, the Cambridge Distributed Computing system and UCLA's Locus file system.

The description present here is not complete; its primary emphasis is on describing the file systems from two points of view: the communications architecture, and the consistency guarantees provided. The communications architecture is crucial in estimating the system's performance. In many systems the communications costs, including operating system's overhead, easily dominate all others; for such systems, analyzing the communications architecture gives a good idea of where bottlenecks will appear in actual operation. Furthermore, the communications architecture includes the client/server interface for the file system. This interface helps determine when the file server machines will become bottlenecks.

The Cambridge Distributed Computing System

The Cambridge distributed computing system [Needham 82] provides file service through the Cambridge file server, an instance of what Birrell and Needham call a "Universal File Server" [Birrell 80].

The file server provides two types of objects, normal **data files**, and **index files**. Index files contain simple directories, and the file server understands their internal structure, at least as far as being able to locate references from an index file to other files. This allows the file server to check the consistency of the file system, ensuring that no index file contains a reference to a non-existent file, and that every file is contained in at least one index file.

The basic operations provided by the Cambridge file server are **open**, **read**, **write** and **close**. The *open* call maps a file name into a temporary identifier (called a **TUID**), which can be thought of as a capability for the file. The *read* and *write* operations read and write data to a file named via a TUID, and the *close* operation takes a TUID and performs finalization processing.

The Cambridge file server actually provides two types of consistency guarantees, one for special files, for which it provides some atomicity guarantees, and one for normal files.

For normal files, read and write operations occur instantly on the file server, giving a very simple, yet useful, consistency guarantee: after a *write* operation completes, a *read* operation anywhere in the distributed environment reads this latest data.

For special files, slightly different guarantees are given. *Writes* to a special file are processed

atomically, at the time the *close* call is made. Before that time, none of the new data is visible to others; afterwards, all of the new data is visible. Optionally, after writing some or all of the data to the new file, the writer of this new data may elect to abort the operation, rather than close the file.

In this case, the new data is discarded, and the file reverts to its state before the transaction started. These atomic transaction semantics are implemented via a shadow page mechanism.

All four of these requests, *open*, *close*, *read* and *write*, are all handled by the file server, and all result in network communication.

Sun's Network File System

The Sun's Network File System (NFS) [Kleiman 86] splits the Unix file system into two levels, the top being the conventional Unix file system interface provided by the Unix system calls, and the bottom being what Sun calls the **vnode** interface (for **virtual inode**). A vnode is intended to be a generalization of the Unix inode, that is, a low-level, efficiently accessed file identifier.

There are several types of vnodes: **files**, **directories** and **symbolic links**. The vnode interface defines operations on these objects.

A typical file system operation decomposes into several vnode-level operations, for example, translating a pathname into its corresponding vnode and applying some vnode-level operation to the resulting vnode. The pathname is evaluated by starting at the root and in turn searching each directory for the name of the next component. The directory search routine is part of the vnode interface; given a directory vnode and a name within that directory, it returns the vnode of the named file or directory. When finished with the entire pathname, the file system has evaluated the corresponding vnode.

Miscellaneous operations, such as *rename*, *delete* and *link*, are applied directly to affected vnodes, or their containing directories.

At first glance, this interface would seem to require a message to the file server for every component of every pathname evaluated. While this cost can be reduced significantly by the use of caching at the client end of the system, a new question arises: how does one ensure the cache contains current information?

NFS handles cache consistency using a simple heuristic: file data is assumed to be valid for 3 seconds after it is fetched from a server; directory data is assumed to be valid for 30 seconds after its arrival.

Such a simple approach has both advantages and disadvantages. On the plus side, NFS is a fairly efficient implementation of a distributed file system. NFS makes extensive use of caching, and this reduces considerably the load an NFS client puts on its file server. And the cache consistency algorithm is easy to implement.

There are two points on the minus side, however. The most significant disadvantage of this approach is that it provides consistency guarantees that are simply too weak for many applications. A program such as a distributed *make* program (a Unix system building tool) would be very difficult to implement under NFS, as the system would have to wait 3 (30 seconds if a directory change is involved), for results to propagate from one machine to another. Such delays could easily make up for any improvement gained from concurrent compilation. This point is discussed in more detail in Section 4.

Another disadvantage to the NFS consistency scheme is, surprisingly, that it still sends too many version-checking messages. Because clients may contact the server every 3 seconds for heavily used files, many messages may be exchanged communicating the fact that the client's cache is still valid. Thus the NFS algorithm polls too infrequently to provide useful guarantees to the applications programmer, yet too frequently to yield minimal server and network loads.

The Apollo Domain System

The Apollo Domain system [Leach 83] uses an interesting interface to the file system: Domain system applications process files by mapping them into virtual memory. After mapping, normal memory reading and writing instructions are used to access the file data.

Objects are identified within the Domain system by 64 bit unique identifiers (UIDs). Associated with each object is a timestamp, indicating when the object was last modified. The basic operations performed by clients of the Domain file system are *read* and *write* operations, whose parameters include the UID and page number requested. *Read* requests return, along with the data, the timestamp associated with the file. If the timestamp indicates that the file has changed since it was first mapped into a process, that process will take an exception on the first reference it makes that is satisfied from the new version. Similarly, a write request sends to the server, along with the data, the timestamp associated with the file before this write request was performed. If this timestamp does not equal the file's timestamp at the server, then an exception is raised in the latest writer's process. The server returns the new timestamp after every write, enabling the client to track the timestamp of files it is writing.

The Apollo Domain system makes extensive use of caching in order to achieve good performance. Apollo has taken a novel approach to cache consistency: the client program is responsible for purging stale data from local caches. To read a file, the application first requests the current timestamp of the file from the server, and then purges all files pages with older timestamps. This guarantees that the application will read the latest data, and will not take any synchronization exceptions, as long as a new writer does not enter the picture while this process is reading

the file. Furthermore, this architecture allows a program to use old cached data, if the application considers old information satisfactory.

Requests to directories are handled somewhat specially in the Apollo Domain system, due to the large number of requests to search directories. Normally, one would obtain a lock on a directory, read the directory and finally release the directory, with each of these three operations resulting in a message being sent to the host storing the data. However, since this operational triplet is so common, the Domain system has an additional call that performs the three operations with one network message. This has no effect on the synchronization guarantees made by their system, except, of course, that it enables them to use their locking architecture for directories without paying an unacceptable cost in additional message traffic.

AT&T Remote File Sharing

The AT&T Remote File Sharing (RFS) system [Rifkin 86] is the prototypical remote system call implementation of a distributed file system. When, in the process of handling a file system system call, the Unix operating system encounters a file system partition exported by a different machine, the entire system call is encapsulated in a message that is transmitted to the partition's real home machine, where it is executed on behalf of the requesting machine.

RFS uses AT&T's STREAMS [Olander 86] network interface, the System V reliable transport networking layer, to provide its communications facilities.

The client/server interface used in RFS is actually the Unix system call interface. For each system call that can encounter a remotely-mounted file system, RFS defines a message that describes the request. When a particular system call encounters a mount point to a remote file system, the entire system call's parameters are encapsulated in this message, and the message is sent to the site actually containing the file. The operation is performed at the remote site, and the appropriate data is returned to the client. In most cases, after the reply arrives, the client has simply to return the error code and any results to the requesting user process.

In 1986, the AT&T RFS system did not do any caching of directory or file data at the various client machines, since remote requests were not handled by the client. Cache consistency was thus not an issue for this version of RFS.

However, more recent versions of RFS [Bach 87] allow a client to satisfy read requests from a local buffer under certain circumstances. Specifically, when one client writes a file that other clients are reading, the readers are notified that they must purge their caches of the appropriate file's blocks. This mechanism operates only while a client has a file open; after closing a file, a client may not re-use data in its buffers without first checking with the server to ensure the

file's version has not changed.

This mechanism is quite similar to that employed in the Andrew file system. However, the Andrew file system essentially keeps track of clients' caches across opens and closes, greatly reducing the number of transactions that involve the file servers. The Andrew file system's cache validation mechanism is described more fully below.

LOCUS

The LOCUS Distributed Architecture [Popek] is one of the most ambitious of the file systems discussed here. Before discussing LOCUS in detail, we must introduce some terminology. Three separate classes of sites are relevant to a file access. The file's **storage sites** (or **SS**) store copies of the file's data. The file's **using sites** (or **US**) are those accessing the data at any one time. Finally, the file's **current synchronization site** (or **CSS**) is a single site responsible for performing various synchronization tasks that must be executed at a single site.

When a file is opened, a LOCUS process sends a message to the file's CSS, in order to perform various synchronization housekeeping chores. This interaction with the CSS results in the US's getting a version number for the file, which it uses in later read and write accesses in order to ensure that the SS it is communicating with has the sufficiently up-to-date information. This is true for directories being searched during pathname evaluation, as well as normal data files. Directory operations are, however, treated specially for operations originating at a SS for the file. In this case, communication with the CSS is skipped, and the process simply accesses the local copy of the data. This algorithm does not lead to the use of out-of-date data because the CSS propagates directory changes to all SS and US machines at the time the directory modification occurs.

Once a file has been opened, LOCUS two types of **tokens** to synchronize operations affecting the open file. Once such token, called a **file offset token**, synchronizes access to file descriptors shared among several processors, while the other token, called a **file data token**, synchronizes accesses to shared inodes. The details two token synchronization algorithms are differ slightly, but essentially work by granting a token that, while possessed, allows the grantee to perform an operation of a particular type, for example, changing an inode. Before a new token granting a particular right can be given out, tokens granting conflicting rights must be rescinded. The essential difference between the file descriptor and the inode token granting algorithms is that file descriptor tokens always grant exclusive access to the file descriptor, while inode tokens may grant either shared (for reading) or exclusive (for writing) access to an inode.

LOCUS does provide some optional support for atomic operations. In particular, one may open a file in a transaction mode, where none of the changes

made to a file are visible until the close system call commits them. LOCUS, like the Cambridge File System, uses an algorithm based upon shadow pages in order to implement these transactions efficiently.

Summary

Most of the systems described above (RFS, LOCUS, and the Cambridge File Server) provide strong synchronization guarantees concerning their use of stale data. In particular, all provide essentially the same guarantees as Unix: as soon as any file system operation completes, the results of the operation are available to any other workstation in the network environment.

On the other hand, all three of these systems communicate with the synchronization site for a file on every file open. This is a potential system bottleneck.

The remaining two systems, Sun's NFS and the Apollo Domain system, do not, by default, provide very strong file system consistency guarantees. In the normal case, a program may, on one machine, make various changes to a file or directory, and programs on other machines will not see these changes until some time later. These systems do, however, make more effective use of caches on the client machines, since the information in these caches may be used without having to explicitly query the server as to the data's validity.

The Andrew file system, described in the next two sections, provides strong consistency guarantees along with the caching efficiencies of the NFS and Domain systems. These sections describe how we did this, and why we bothered.

The Goals of the Andrew file system

The Andrew file system was designed to provide the appearance a single, unified file system to a collection of approximately 5000 client workstations connected via a local area network. In the briefest terms, the Andrew file system must be fast and inexpensive. With so many workstations connected to the file system, the file system must not become a new bottleneck, now that adequate processor cycles are available.

Furthermore, the Andrew system is supposed to provide inexpensive computation resources for students at a University. The workstation cost is expected to be eventually less than a few thousand dollars, and the incremental cost of adding a client to the file system must be considerably less than the cost of the workstation itself. Our goal was that a single file server should be able to provide service to approximately 50 clients. We assume that a file server is a normal workstation augmented with approximately 1 gigabyte of disk storage.

Since the Andrew file system is intended primarily for use by Unix workstations, it must provide enough functionality to support the Unix file system.

The Unix file system does not provide any support for stable storage except for a system call that guarantees a file has been completely written to the disk. Similarly, Unix does not support any sort of transactional facility. Thus the Andrew file system need not, and does not, provide any specialized transactional facilities.

On the other hand, the Andrew file system does provide useful consistency guarantees. Foremost in this regard is the guarantee that after the completion of an operation, the next operation performed anywhere in the network will see the updated file system state. For instance, after a rename system call completes on one machine, no client on the network is able to open the file under its old name, and all are able to open the file under its new name. Similarly, after a file being written is closed, any client on the network opening that file reads the new contents of the file, never the old.

In short, with one exception, discussed next, the Andrew file system guarantees that after a file system call completes, the resulting file system state is immediately visible everywhere in the network.

To understand this exception, one must understand how programs read or write files under Unix. A Unix process examining or changing the contents of a file first opens the file, specifying whether the file will be modified. Next, the process performs a sequence of read and write system calls (the latter only if the file was opened for writing). Finally the process closes the file.

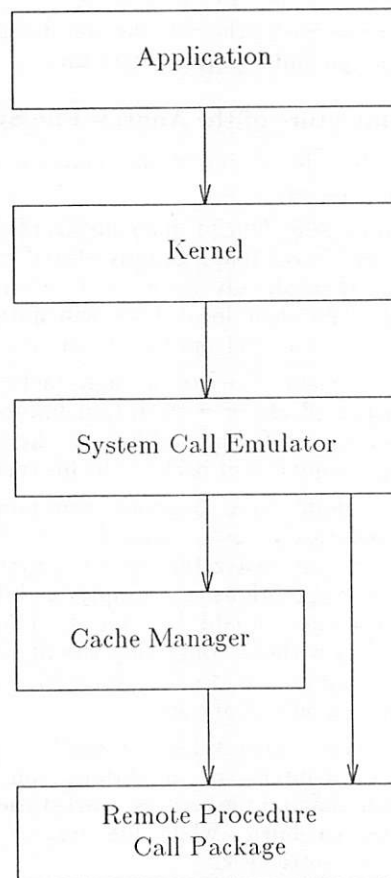
On a standard Unix system, after the completion of any write system call, the data written by that call will be read by any process reading from the same file. This is true whether or not the reader opened the file before or after the writer performed the write system call. Thus two different users of the same file see changes made to the file appear at the read and write system call granularity.

In the Andrew file system, however, the new data written to a file is not actually stored back at the file server until the file is closed, and data cached on the workstation is only checked for currency at the time a file is opened. Thus, when sharing a file, users of one workstation will not see the data written at another workstation until the first open system call executed after the writer closes the file.

We chose these semantics for very pragmatic reasons. First, very few programs are disturbed by seeing changes at an open/close system call granularity rather than at the individual read/write system call granularity. Actually, few programs are even prepared for the possibility of a file's changing while they are processing it. In addition, our kernel modifications do not allow us to intercept read and write system calls to files located in the Andrew file system, a restriction imposed for efficiency reasons. Finally, when most programs finish processing a file, they exit, closing all of their open files. Thus Unix

generally sees a close system call issued with respect to a file when a program is finished processing that file.

In actual practice, using the open/close granularity for consistency guarantees instead of the read/write granularity has not caused us any troubles.



Of course, the above discussion explains why our consistency semantics are strong enough; it does not demonstrate that they are necessary. There are several reasons we chose to make these consistency guarantees. First, many people in our environment use several workstations simultaneously, using a window manager to maintain several windows on multiple machines. These users can effectively move from one machine to another simply by moving the cursor from one window to another. For the system to be comprehensible, it is important that when a program in one window announces that it has done something, the results are immediately available to programs running in any of the other windows.

We considered requiring users to explicitly request the latest version of a particular file, rather than guaranteeing that the file system simply provides that version. We rejected this alternative, since it would burden naive users with a considerably more complex system model. Rather than thinking of a file as containing certain data, independent of the workstation from which it is viewed, the user would have to

understand how data propagates in our system.

This can turn out to be a considerable intellectual burden. After using a remote processor to compile and install a new remote procedure call package, for example, a user should not be required to explicitly invalidate all of the object files, header files, and documentation that had just been installed.

Given these goals, the next section describes our file system design and how it achieves them.

The Architecture of the Andrew File System

The Andrew file system design embodies a small set of design principles.

Workstation caching Our primary means of minimizing file server load. Simply stated, workstations that already have most of their data cached on their local disks will not have to retrieve as much from the file servers.

Write-through cache A write-through cache minimizes the effects of workstation failure. Thus every time a file is modified on the workstation, a copy is sent back to the file server.

Minimize communication Network communications is moderately costly, and the less of it performed, the faster the resulting system will be. For operations that simply read data, the workstation should be able to handle the requests without contacting the file server at all, assuming the proper information is in the workstation's local cache.

Minimize server load When possible, perform processor-intensive operations on clients rather than on file servers; workstations have cycles to burn, while file servers quickly become bottlenecks.

Consistency We wanted our file system to provide the consistency model described above.

These five design principles guided our entire design.

The Andrew file system consists of two types of machines, **clients** and **servers**. Clients run a program called **venus**, which essentially connects the workstation to the collection of file servers. Venus receives requests from the Unix kernel in the form of intercepted system calls, and translates them into operations on the workstation's local cache, as well as remote procedure calls to one or more file servers. Occasionally, **venus** receives RPC requests from a file server, notifying it that a callback, described below, has been broken.

Files and File Identifiers

Files are named at the **vice/venus** interface by 96 bit tokens called **fids**, for **file identifier**. Unlike some systems, such as Accent's Sesame file system [Jones 82], the Andrew system's fids do not name immutable objects: the contents of a file named by a single fid can have different contents at different times.

Files are comprised of a **data** part and a **status**

part. The data part of a file consists of a stream of 8 bit bytes. The status part of a file contains various status information concerning the file, such as its length, the date the file was created, and the protection information associated with the file. One important piece of status information is called the **data version number**. A file's data version is incremented every time the file's data is changed, and is used by many of our consistency algorithms. The workstation caches the data part of a file and the status part of a file independently.

A file's fid is sufficient to retrieve the file's status or data parts from a file server at any time. Because files are not immutable, this file may contain different data at different times. However, the combination of a file's fid and its data version do uniquely specify a file's contents.

Files are stored on one or more file servers. Any file server can, when presented with a fid, return a list of those servers currently holding the corresponding file.

Directories and Vnodes

The Andrew file system can be viewed from two levels, the lower level being a flat name space, where each file is identified by its *fid*, and the higher level being a hierarchical name space where files are identified by a pathname.

In designing the Andrew file system, we were concerned about the processor time required to translate pathnames to *fids*. We were especially concerned that file servers would not have enough processing resources available to handle extensive pathname-based computations. For this reason, we have tried to keep all knowledge of pathnames, as well as all requests that deal with pathnames, out of the file server interface. Instead, operations that deal with pathnames are handled by the workstation's *venus* process, with *venus* decomposing these operations into a series of operations at the *fid* level.

In more detail, the cache manager provides a useful abstraction: fast access to the data and/or status parts of files named by *fids*. The *venus* system call emulator is built on top of the cache manager, and invokes the cache manager when it needs to fetch the status or contents of a file. For instance, when translating a pathname into a *fid*, *venus* begins by fetching the latest version of the root directory (whose *fid* is determined at bootstrap time) into the cache. This directory is searched for the next component of the pathname, and this new file's *fid* is extracted. If this is the last component of the pathname, *venus* has now completely evaluated the pathname into a *fid* that can be passed either to the cache manager or, via the remote procedure call interface, to a file server. If this is not the last component of the pathname, *venus* iterates, searching this new directory for the next component of the pathname.

After the pathname has been reduced to a *fid* by the system call emulator, the emulator performs the

requested operation on the *fid*. For example, if this is an *open* system call, *venus* will fetch the data part of the file into the cache, and arrange for further read and write operations to be directed to the file in the cache. If *venus* is emulating an *chmod* system call, it will issue a remote procedure call of the procedure *ViceStore*, with the *fid* of the target file, and the updated file status.

Cache consistency, in this architecture, amounts to ensuring that calls to the cache manager return up-to-date data from the cache. Rather than polling the file server on each call to the cache manager, or simply assuming that the data in the cache is up-to-date, the Andrew file system uses some state shared between the file servers and the client workstations to notify client workstations when their caches contain invalid information. This shared state is called the

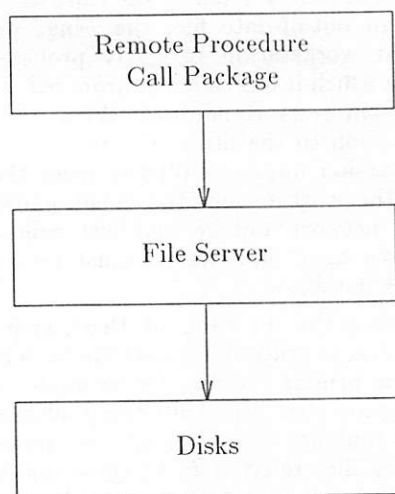


Figure 2. File Server Structure

callback database, and is described in the next section.

Callbacks

The Andrew file system maintains cache consistency via **callbacks**. Simply, when an Andrew file system client fetches a file's data or status information from a file server, it may also receive a promise from the file server that the file server will notify the workstation before it changes the data or status information associated with the file. Such a promise is called a **callback**. The process of notifying the appropriate workstations when a file has changed, which must be performed before changing the contents or status information associated with a file, is called **breaking a callback**.

Note that when a workstation has a callback outstanding on a file in its cache, it can read the data or status associated with that file from its cache without any fear that the information is out-of-date. This considerably reduces the load that a workstation places on a file server, without weakening the cache

consistency guarantees.

Note that while callbacks are essentially a simple concept, their implementation has some tricky aspects. These problems arise in two areas: synchronization of calls to the file server with callback breaking messages from the same file server, and communications failure while breaking a callback promise.

Callback Interface

This section describes how callback promises are made and broken via the client/server interface in the Andrew file system. In the Andrew file system, both the clients and the file servers act as remote procedure call servers. The file servers export the standard file server interface, while the clients export a simpler interface allowing file servers to break callbacks.

Only three calls in the file server interface involve callbacks in any way. The *ViceCreate* call creates a new file in a directory. It returns the *fid* of the newly-created file, and this *fid* is returned along with a callback promise. The *ViceFetch* call is used to fetch the status, or status and data parts of a file from the file server, and also implicitly returns a callback promise for the file involved. Finally, the *ViceDelete-Callback* call notifies the file server that the client is no longer requires its callback promise.

The callback interface exports several procedures for the file servers. The most important, of course, is *Callback*, which directs the client to break the callback promise associated with a particular *fid*. There is also a procedure allowing the file server to break all callback promises associated with an entire set of files, for use when those files are moved from one file server to another.

Note that the only parameter passed from the file server to the client when a callback promise is broken is the *fid* of the associated file. We discovered that more information is required to avoid certain race conditions; this problem is discussed in more detail in the next section.

Callback Synchronization

The problems involved in handling callback-breaking messages in *venus* all occur when an outgoing to the file server returns a callback promise at the same time that the file server calls the client to break a callback promise associated with the same file. Due to the lack of precision in the client/server interface, *venus* can not tell if the callback-breaking message applies to the just-returned callback promise, or to a previous one, and thus *venus* does not know if it actually has a callback promise on the particular file.

There are several possible solutions to this problem.

One's first reaction when faced with synchronization problems of this nature is to consider locking the appropriate data structures. The structures involved in this case are the cache entries in the client, and one could lock cache entries when making requests that may return a callback promise, or when processing a

callback-breaking message. The problem with this proposal is that it is very difficult to determine a valid locking hierarchy to avoid deadlock; we were unable to.

It is easy to understand the difficulty: most file system operations require locking the client's cache entry before making a call to the file server. Yet the file server is the only site with the knowledge of which callback promises must be broken during the operation, and thus which cache entries must be locked at other clients. The file server may discover, while processing a request, that the locking partial order has already been violated by the client initiating the request.

Unfortunately, in the situation where the file server discovers the locking hierarchy has been violated, the file server has no recourse but to break some client lock. And allowing for this possibility greatly increases the complexity of the client's code. Thus we ruled out the locking proposal as infeasible.

Another proposal was to add a version number to the callback database at the file server. All changes to the callback database are generated by the file server, thus the file server can order changes to the callback database, and mark all messages with an indication of which version of the callback database the messages refers. When a message arrives breaking a callback promise, while a remote procedure call is being processed that eventually returns a new callback promise, the two operations can be ordered at the workstation by simply comparing the callback database sequence numbers and processing them in the same order as the server.

An ad hoc solution is also possible, based upon the observation that a client may always discard a callback promise without affecting correctness. When a callback promise is returned by a call that overlapped the reception of a message breaking a callback promise for the same *fid*, *venus* can simply discard the callback promise.

While the callback sequence number solution is by far the best, we actually have implemented the ad hoc solution instead, since we discovered this problem while actually running the system, and fixed it with the fastest emergency patch we could design. Our estimate is that only a few callback promises are incorrectly discarded per week in an environment containing several hundred workstations, so we have never taken them time to implement the cleaner solution; rather we opted to extensively comment the ad hoc one.

Callback Failure

The callback algorithms described above depends upon the reliable delivery of messages to break callback promises. Temporary network outages can prevent the delivery of callback breaking messages; the ability to tolerate these outages is important.

One of the more virulent manifestations of this problem occurred with our printer spooling software,

although many systems are vulnerable to these failures. In the Andrew file system, a workstation can run for an unbounded amount of time between calls to the fileserver, because the presence of a callback promise permits the client workstation to treat its locally cached information as correct. If a network failure occurs while delivering a callback-breaking message from a file server, then from the time the failure occurs until the time the client tries to contact the file server again, the workstation will not receive callback breaking messages. During this time, the client incorrectly believes it has the current version of some files. As soon as the client either successfully contacts the server, or fails to due to network troubles, *venus* will realize that it has not been receiving callback breaking messages.

To bound the time that a workstation can run with an out-of-date version of the callback database, and thus an out-of-date file, the *venus* process on every client workstation regularly probes each file server from which it has callback promises. Currently probes are sent every 10 minutes. When a probe from the workstation to the file server returns a callback database version number different from the version stored at the workstation, the client knows that a temporary network outage had lost some callback breaking messages, and that it must re-synchronize the callback database.

Bounding the duration of these error-induced inconsistencies is crucially important to certain programs. Our printer spooler, for example, is a very simple program that wakes up every 30 seconds and checks the contents of a directory for queue entries, printing any files referred to in those queue entries. When done printing a file, the spooler deletes the queue entry for the request.

Under normal operation, the printer spooler machine has a callback promise on the spooling directory, and the checks it performs every 30 seconds does not result in communication with the file server. When this callback promise is broken, however, the next time the spooler scans the directory, it fetches the new contents of the spooling directory. Were the callback breaking message lost due to a network outage, the printer spooler would never see a change to the spooling directory again. The addition of the periodic polling on behalf of the workstation means that for the printer spooler, a temporary network failure leads at most to a ten minute suspension of services after network operation has been restored.

Callbacks in Other Systems

This section describes how callbacks could be used to implement the stricter consistency guarantees made by RFS and other distributed file systems. As RFS provides the strongest guarantees of any file system discussed here, the remainder of this section discusses only RFS.

There are two areas where the Andrew file

system does not provide consistency guarantees as strong as those provided by RFS. The first area is that of granularity: RFS guarantees that a read system call anywhere in the system always reads the latest written data, whether or not the file has been closed. The second area is locking: the Andrew file system does not support locking very efficiently--each lock call sends a message to the appropriate file server. In an environment where open calls implicitly set shared or exclusive locks, these extra messages would eliminate any advantages obtained from callbacks.

Our problem of checking the cache for stale data on with read/write system call granularity is more an artifact of our implementation than a fundamental design choice required by the callback architecture. For performance reasons, the Andrew file system does not intercept read and write system calls, and so can not perform cache validity checks upon their execution. In a kernel implementation, the Andrew file system could perform the same validity checks upon every *read* that it does upon every *open*, producing the same consistency guarantees at the finer *read/write* system call granularity.

The more interesting issue is the implementation of a correct locking protocol with minimal message traffic. In an environment where one can open a file, read some data and close that file, all without any network traffic, one would hope that setting and clearing the appropriate locks could also be done without network traffic.

This can be done fairly simply by a form of "lazy-evaluation." When a process releases a lock, it need not actually communicate with the file server at the time. Instead, the workstation itself continues to hold the lock, assuming that a similar lock request may be forthcoming. If a process on the same workstation requests the lock, it can be immediately granted. If a process on another workstation requests an incompatible lock, the file server must first request the workstation give up the lock. As long as no process actually holds the lock at the time, the client can do this immediately; otherwise, the client must delay granting the file server's request until the user process is done.

In short, network traffic is required for obtaining a lock only when an incompatible lock was previously granted to a client on another workstation.

Conclusions

We have learned a great deal from our implementation of the Andrew file system, and many of these lessons can be applied in building any other distributed file system.

Of course, caching is critical for getting good performance from a distributed file system. Caching is also a very powerful tool for reducing the load a fixed number of workstations will place on a file server.

Moreover, powerful cache consistency guarantees are very useful for casual users of the system, as well as for programmers engaged in building applications. While these guarantees are not trivial to provide, they are well worth the effort. Even relatively naive users of our system use the Butler system [Nichols 87] to commands on otherwise idle machines. These users have come to expect that when their status monitoring program announces new mail has arrived, they will be able to read the new mail with the mail-reading program, without regard to the location of either of these programs. Our system maintenance procedures involve compiling programs on several machines (and machine types) concurrently; building the procedures to synchronize these processes was greatly simplified by our relatively strict file system consistency guarantees. Students have even written multi-user games using the Andrew file system to effectively provide shared memory (this application put a considerable load on the file server). And, of course, people writing and debugging distributed systems especially appreciate the convenience good consistency guarantees provide. In short, strong consistency guarantees are useful for many classes of users, from the most naive to the most sophisticated.

Finally, and most importantly, an efficient file system providing powerful consistency guarantees and which makes extensive use of caching can be built, using *callbacks* for synchronizing changes to files stored in workstations' caches. The use of *callbacks* in the Andrew file system handles between 97 and 99 percent of the requests that otherwise would have to contact the file server to ensure cache consistency. The Andrew file system is one example of such a file system currently in use by over 300 workstations on the Carnegie-Mellon University campus.

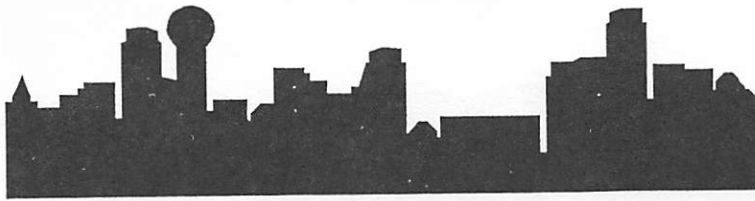
It is important to note that the callback architecture used in the Andrew file system is not essentially incompatible with the numerous distributed file systems surveyed in Section 2 of this paper. For example, the interface provided by NFS file servers is quite similar to that provided by Andrew file servers, and we believe that NFS could be extended to make similar use of a callback architecture. Remote open-style file systems, such as AT&T's RFS, could also use callbacks to reduce network traffic.

In summary, callbacks provide a tool applicable to a variety of distributed file systems, enabling a distributed file system to make strict guarantees as to the currency of cache information without requiring explicit communication with a common synchronization point at every reference. The use of callbacks in a distributed system allows it to provide both good performance and useful consistency guarantees.

References

- [1] M. J. Bach et al., A Remote-File Cache for RFS, *Usenix 1987 Summer Conference Proceedings*: 273-279, June 1987 Usenix Association, P.O. Box 7, El Cerrito, CA.

- [2] A. D. Birrell and R. M. Needham, A Universal File Server, *IEEE Transactions on Software Engineering*, New York, Se-6(5), 450-453, Sept. 1980.
- [3] A. D. Birrell and B. J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computing Systems*, 2(1): 39-59, Feb. 1984.
- [4] M. B. Jones, R. F. Rashid, M. Thompson, *Sesame: The Spice File System*, Internal Documentation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh PA, 1982.
- [5] S. R. Kleiman, Vnodes: An Architecture for Multiple File System Types in Sun UNIX, *Usenix 1986 Summer Conference Proceedings*: 238-247, June 1986, Usenix Association, P.O. Box 7, El Cerrito, CA.
- [6] H. F. Korth, Locking Primitives in a Database System, *Journal of the Association for Computing Machinery*, 30(1): 55-80, Jan. 1983.
- [7] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson and Bernard L. Stumpf, The Architecture of an Integrated Local Network, *IEEE Journal on Selected Areas in Communications*, Vol. SAC-1 (5): 842-857, Nov 83.
- [8] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal and F. Donelson Smith, Andrew: A distributed Personal Computing Environment, *Communications of the ACM* 29 (3): 185-201, Mar. 1986.
- [9] R. M. Needham and A. J. Herbert, *The Cambridge Distributed Computing System*, Addison-Wesley Publishing Company, 1982.
- [10] B. J. Nelson, *Remote Procedure Call*, Ph.D thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh PA, May 1981.
- [11] David A. Nichols, Using Idle Workstations in a Shared Computing Environment, to appear in the *Proceedings of the 11'th ACM Symposium on Operating Systems Principles*, November, 1987.
- [12] D. J. Olander, et al., A Framework for Networking in System V, *USENIX Conference Proceedings*: 38-46, Atlanta, Georgia, June 1986.
- [13] Gerald J. Popek and Bruce J. Walker, *The LOCUS Distributed System Architecture*, The MIT Press, Cambridge, Massachusetts.
- [14] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, Kang Yueh, RFS Architectural Overview, *Usenix 1986 Summer Conference Proceedings*: 248-259, June, 1986, Usenix Association, P.O. Box 7, El Cerrito, CA.
- [15] D. M. Ritchie and K. Thompson, The Unix time-sharing system, *Bell Systems Technical Journal* 57(6): Jul.-Aug. 1978.
- [16] R. Rodriguez, M. Koehler, R. Hyde, The Generic File System, *Usenix 1986 Summer Conference Proceedings*: 260-269, June, 1986, Usenix Association, P.O. Box 7, El Cerrito, CA.
- [17] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector and Michael J. West. The ITC Distributed File System: Principles and Design, *Proceedings of the Tenth ACM Symposium on Distributed Systems Principles*, 19(5): 35-50, Dec., 1985.



USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

A Multi-media Message System for Andrew

Nathaniel Borenstein
Craig Everhart
Jonathan Rosenberg
Adam Stoller
Information Technology Center
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

ABSTRACT

The Andrew Message System (AMS) is a portable, distributed, multi-media, multi-interface system for reading and sending mail and bulletin board (bboard) messages. Mail and bulletin board processing was selected as a "showcase" application to demonstrate how the Andrew file system and user interface toolkit could be applied most usefully to a user's daily tasks.

The AMS supports multi-media messages, which may include line drawings, hierarchical drawings, spreadsheets, raster images, animations, and equations. It is explicitly designed to support a huge database of messages and an enormous user community. At CMU, it services over 1200 bboards, including netnews, the Dow Jones information service broadtape, and bboards on which newspaper cartoons appear as raster images. The system incorporates a B-tree based "white pages" for doing name lookups, including phonetic matching of misspelled names. In addition, the system supports a number of advanced features such as voting on multiple-choice questions, private bboards, shared mailboxes, and automatic classification of incoming mail messages. The server-based architecture makes it easy for client interfaces to be ported to or built on almost any computer. Currently, interfaces run on IBM RTs, DEC MicroVaxes, Suns, IBM PC's, Macintoshes, and Vax UNIX and VMS timesharing systems.

Introduction

In the Andrew project [1], the primary focus of the development effort has been on developing two major tools to support an advanced computing environment. Those tools, the Vice distributed file system and the Andrew Toolkit (a.k.a. BE2) for user interface development, are described in detail elsewhere [2,3]. In addition to developing these primary tools, the Andrew project has created the Andrew Message System (AMS), an application suite that uses the Andrew tools to facilitate electronic communication by users of the system. As a large application developed independently of (but in cooperation with) the BE2 and Vice projects, the AMS has been both a useful application in its own right and a driving force in the evolution and improvement of Vice and BE2.

The goals of this paper are twofold: we will describe the key features and architecture of the Andrew Message System, and we will explain how the underlying Andrew facilities simplified the development of such a system and made it more powerful and useful than would otherwise have been possible.

The Goal: A Truly Integrated Message System

The primary goal of the Andrew Message System project was to develop an *integrated* electronic mail and bulletin board system that took advantage of the primary Andrew facilities. The word "integrated" has at least four meanings in this context:

Integration of functionality. The system should integrate many advanced features already present in other systems into a reliable and efficient unified system.

Integration of media. It should be possible to send a collage of anything that can be digitized through the mail, in a seamless and painless manner.

Integration of disparate hardware. The system should be useful not only on advanced workstations, but also on IBM PC's, Macintoshes, and via terminal/dialup access to time-sharing systems based on several operating systems.

Integration of database access. The system should provide an integrated database access mechanism, with as much as possible of the mechanism encapsulated in interface-independent libraries, so that the construction of new user interfaces to the

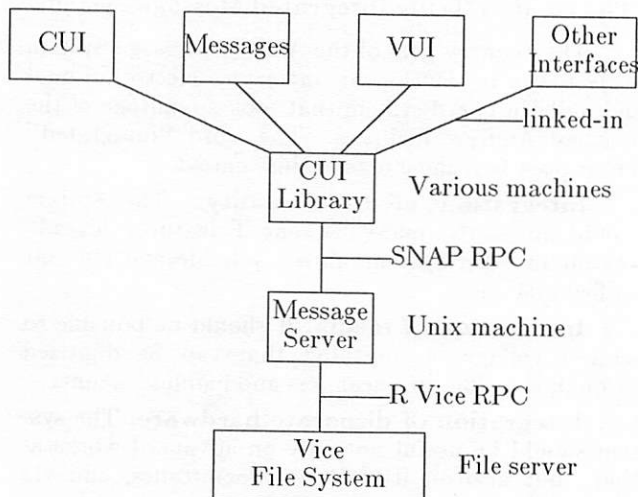
system will be relatively quick and painless.

Design Architecture

The integration goals described above led to a system architecture in which functionality is divided into three levels:

Actual access to the message database occurs exclusively at the *message server* level. The message server is a process that runs on a machine with full access to the Vice file system. Multiple client programs may simultaneously access data from the same message server, via a remote procedure call mechanism known as SNAP.

Various abstractions relevant to client interfaces, as well as the interfaces that allow communication with the message server, are implemented at the level of the *client library*, generally known as the CUI (common user interface) library. This library is written in portable C, and runs in environments as diverse as UNIX, VMS, MS-DOS, and Macintosh. Finally, interaction with the user occurs at the *application level*, which is, of course, different for each user interface. Some of the user interfaces are highly portable across environments, while others are not. For example, the most basic user interface, CUI (Common User Interface) will run in every environment where the CUI library runs, and requires no more sophisticated a display than a teletype. A more popular user interface, VUI (Visual User Interface) runs on PC's, and on UNIX and VMS systems with CRT displays. Other more specialized interfaces include Messages (formerly known as AUI, the Andrew User Interface), which runs on a UNIX workstation running X11 or the old Andrew window manger, and MacMessages, which runs only on a Macintosh. The levels of the system are pictured below:



Of course, this description leaves out many important features. Most notably, it leaves out a complex message delivery system, developed as an alternative to the UCB sendmail program for reliable mail delivery in the Vice environment. Although the delivery system component of the Andrew Message

System will be mentioned briefly below, a full description of it is beyond the scope of this paper [5].

Features

In this section, we outline some of the key features of the Andrew Message System.

Multi-media Messages

The most noticeable feature of the Andrew Message System is its support for a rich set of multi-media objects. In particular, the AMS allows users to send, in addition to normal text messages, formatted text, raster images, line drawings, hierarchical drawings, interactive tables, equations, and even animations. The multi-media objects are managed by the Andrew BE2 toolkit, as described elsewhere [2].

Of course, the multiple machine architecture makes multi-media mail particularly challenging. In the AMS, only one user interface, Messages, is currently able to display the full complement of multi-media messages. When other interfaces – notably those that run on lower-functionality hardware – display multi-media messages, they ask the message server to give them an “unformatted” version of those messages. Thus, where a Messages user might see something like this:

Date: Wed, 25 Nov 87 10:52:30 -0500 (EST)

From: Nathaniel Borenstein <nsb+@andrew.cmu.edu>

Subject: Animated logo!

Here's the CMU logo:



But the *best* part is that if you click on it and choose the **Animate** menu, it will turn into a cube and tumble around on your screen!

Isn't that neat?

A user of a lower-functionality interface would instead see something like this:

Date: Wed, 25 Nov 87 10:52:30 -0500 (EST)

From: N. Borenstein <nsb+@andrew.cmu.edu>

Subject: Animated logo!

Here's the CMU logo:

[An Andrew/BE2 view (an animated drawing) was included here, but could not be displayed.]

But the best part is that if you click on it and choose the Animate menu, it will turn into a cube and tumble around on your screen!

Isn't that neat?

Note that all of the text is still presented correctly, and the full multi-media message persists in the database, so that the frustrated user may seek out an Andrew workstation if he wishes to view the message in its full multi-media splendor.

Multiple Machine Types & Interfaces

As previously indicated, one of the major design goals of the AMS was to create an integrated electronic communication environment unifying a wide variety of computer hardware. That this goal has largely been achieved is indicated by a list of the available AMS interfaces available and the hardware/software environments in which they run (current as of winter, 1987/88):

Messages (*IBM RT, Sun, MicroVax*) – The flagship interface, supporting full multi-media messages.

CUI (*IBM RT, Sun, MicroVax, IBM PC, Macintosh, VAX UNIX or VMS timesharing system*) – the simplest, teletype-oriented interface.

VUI (*IBM RT, Sun, MicroVax, IBM PC, VAX UNIX or VMS timesharing system*) – a screen-oriented interface with a interface style like that of many IBM PC applications

MacMessages (*Macintosh*) – a native Macintosh interface.

Batmail (*UNIX/EMACS*) – a user-supported EMACS interface.

Integration of Message Types

Another goal of the AMS was the integration of various kinds of electronic communication in a common system. The synergistic effects of merging mail and bulletin boards in a common interface are especially beneficial. For example, if a user sees an interesting message on a bulletin board and wants to keep a copy, he may simply put it into one of his mail folders, the same way that he would manipulate his own mail.

In addition, the need to explicitly address multiple protection domains within the message database – to allow for public bboards and private mail in the same system – has enabled the creation of various “hybrid” forms of message databases. For example, there are private bulletin boards shared by groups of users and official bulletin boards to which only certain users may post messages but which everyone may read. Users may even give each other access to their mail folders so that, for example, a secretary may process his employer’s mail as if it is a bulletin board he administers.

Rewritten Delivery System

Originally, we had hoped to build the AMS without becoming entangled with issues of message delivery. In particular, we had hoped to be able to simply use the standard Berkeley sendmail program as our delivery engine, and concentrate our efforts on other things. Although the AMS will, in fact, run

compatibly with a standard sendmail-based delivery system, that system is simply not reliable in the Vice environment, nor would it be likely to be reliable in any similar environment.

Because *sendmail* was not written with the notion that the file system could temporarily disappear (for example, during network or file server problems), it is very hard to make it behave consistently and reliably when such situations occur. Additionally, sendmail takes as a given the notion that users live on particular machines, whereas, in the Vice environment, it is basically irrelevant what machine the recipient uses, as long as that machine is connected to Vice. Sendmail also depends on some features of standard UNIX not available in the authenticated Vice environment, particularly those related to file protection and special user privileges. Finally, the Vice file system provides a much greater level of authentication than standard UNIX, and sendmail was not well set up to take proper advantage of that authentication information.

For all of the above reasons, an entirely new message delivery system was implemented as part of the AMS. That delivery system will be described in detail in a forthcoming paper; a full description is beyond the scope of this paper.

The delivery system is technically an optional component, in that the AMS can be (and often is) configured to run with the standard delivery system on non-Vice machines. However, several features of the AMS work only in the presence of the AMS delivery system. In particular, the AMS delivery system allows us to make very strong guarantees about message authentication, whereas it is notoriously easy to forge messages on a standard UNIX system. The AMS delivery system also provides a great deal of location-independence; users need not send their mail to a particular machine, but merely to an address that corresponds to an entire Vice installation. Finally, the AMS delivery system provides sophisticated user name lookup features via a mechanism known as the *white pages*.

White Pages

The white pages database is a package that facilitates the mapping of human names to UNIX user names and other mail-related information. It is optimized for efficient operation in a distributed environment by being packaged in a collection of Vice files are organized according to a B-tree discipline [6]. The white pages facility, which features heuristics for phonetic matching of user names, is used primarily for evaluating and rewriting what users type on the “To:” and “CC:” lines of outgoing messages, and for evaluating the addresses on incoming messages from remote sites. The operation of the white pages is most simply demonstrated, however, by a sample dialog with the CUI program, using the “whois” command to probe the white pages database directly. (The user input is in **bold font**.)


```

CUI> whois bond
Verifying name list 'bond'...
The name 'bond' is ambiguous.
What did you mean by 'bond'?
1 - Eric Bond (eng) <bond+@andrew.cmu.edu>
2 - Sandra Bond (itc) <sandra+@andrew.cmu.edu>
3 - None of the Above
Choose one [3 - None of the Above]: 2

Sandra Bond <sandra+@andrew.cmu.edu>
CUI> whois not bronstin
Verifying name list 'not bronstin'...
Name match was uncertain; please confirm:
1 - Nathaniel Borenstein <nsb+@andrew.cmu.edu>
2 - None of the above
Choose one [2 - None of the above]: 1

Nathaniel Borenstein <nsb+@andrew.cmu.edu>
CUI> whois AMS
Verifying name list 'AMS'...
The name 'AMS' is ambiguous.
What did you mean by 'AMS'?
1 - Anthony Iams (cdec) <ai02+@andrew.cmu.edu>
2 - Wendi Anne Amos (csw) <wa0a+@andrew.cmu.edu>
3 - None of the Above
Choose one [3 - None of the Above]:

AMS (Address Validation Error: ambiguous name)
CUI>

```

Because the white pages database is stored in Vice, it is accessed by the message server, which then exports this function to the client interfaces, wherever they may be. Thus the above dialog could take place as easily on a PC as on a Vice-based workstation. The White pages is also used by the Andrew *finger* service.

Miscellaneous Features

In addition to the major areas described above, there are several smaller but still interesting features implemented in the Andrew Message System, some of which will be briefly summarized here.

Magazines. In order to help cope with the flood of information produced by large bulletin board systems, the AMS permits the creation of magazines. Magazines are bulletin boards that volunteers from the user community maintain on topics of their own choosing. For example, the editor of a magazine on "music" might read twenty music-related bboards at a site like Carnegie-Mellon. He can then easily cross-post the messages he considers worth reading on his magazine. Others, lacking the time or inclination to read all twenty of those bboards, can instead read only the magazine, thus seeing only the "cream of the crop".

Automatic Classification of New Messages. When the message server processes new incoming messages from a user's mailbox, by default it simply puts them into a folder called "mail". However, the user may supply a program which the message server will

run on each piece of incoming mail to decide where that message should be classified. Currently, the program must be written in a somewhat cumbersome stack-oriented language, although this language is slated for replacement in a future release (see "future plans", below). Although clumsy, the language is powerful enough to permit, for example, hundreds of Internet mailing lists to be sent to the same mail address and then be automatically sorted onto appropriate bulletin boards on the basis of information in the message header.

Voting. AMS users can compose special messages that call for "votes". Anyone reading such messages is automatically prompted (by the CUI library) to vote on the issue in question. Votes are automatically mailed back to an address specified by the vote composer, and there are also a few features to make vote tabulation straightforward. Voting is *not* by secret ballot - indeed, with the AMS delivery system, the votes are authenticated, so that multiple voting is easily detected. All votes are multiple-choice, although the vote composer may specify whether or not write-in votes are permitted.

Return-receipt requests. On sending a message, an AMS user can designate it as requesting an acknowledgment, or return-receipt. When an AMS user receives such a message, he is automatically asked whether he wishes to send such an acknowledgment. This is particularly useful for users who are separated by unreliable network connections.

Enclosures. Often, users at distantly-connected sites need to share files or other data, and the only means at their disposal is electronic mail. In such cases, the "enclosures" feature of the AMS can be useful. With this feature, a user can specify that all or part of his message is an "enclosure" to be handled specially. When a user receives such a message, he is automatically asked whether he would like to do something special with it, such as write it to a file or (on UNIX) pipe it through a process. Of course, what is written to the file or pipe is the enclosure only, without the message headers, which can save a fair amount of time in processing such messages.

Folder creation announcements. Because the AMS makes it very easy for users to create new bulletin boards, the process of subscribing to new bboards must be very simple. One mechanism that supports this is folder creation announcements. When a user reads a message that is a folder creation announcement, he is automatically asked whether or not he wishes to subscribe to the newly created folder, or bulletin board. Such announcements are typically created automatically when the system creates a new bboard, but may also be created by users themselves.

Assessment of the Andrew Environment

The AMS represents a great deal of development effort by the AMS developers themselves, independent of the work of the Vice and BE2 development efforts.

However, the AMS would not be nearly the system it is without the work of those other groups. In this section, we will describe how the AMS benefits from the more general Andrew facilities, and what price had to be paid for those benefits.

The Vice File System

The AMS as we know it would not be possible without the Vice file system. The AMS was designed specifically to handle enormous message databases (i.e., gigabytes of message text), which would simply be impractical to store on individual workstations. In addition to the support for the scale of the system, Vice also gave us much stronger authentication and richer file protection than does the normal UNIX file system. As a result, in the Vice environment we can make a relatively strong guarantee to our users that mail from other local users is not forged, and we can easily permit the creation of private bulletin boards and the sharing of mailboxes.

The most important benefit of Vice for our purposes, however, was the simplification of the database. Rather than having to replicate the database on multiple machines, we can simply store it in one place – Vice – and let all the interested message server processes read the data from that place. This was a tremendous simplification over any scheme for sharing bulletin boards among multiple machines that do *not* share a common file system.

On the other hand, Vice poses a few notable problems for developers, when contrasted with normal UNIX. Most notably, Vice does not support setuid programs, hard links between files not in the same directory, or per-file protection information (Vice protection is on a per-directory basis). The absence of these features placed additional constraints on the design of some of our algorithms.

Perhaps most important, the use of Vice *or any other distributed file system* is likely to seriously diminish the reliability of applications that were written for a non-distributed file system. In particular, most pre-existing UNIX utilities have no idea that they should check the return value of the `close()` system call, nor of what they should do if such calls fail. (The failure of a `close()` call is almost inconceivable on the standard UNIX file system, but is a common point of failure on Vice. Such failures may correspond to file server failures, or to other problems, notably related to file protection and storage quotas.) This kind of incompatibility can necessitate massive rewrites of pre-existing software; in our case, it was a major factor in our decision to rewrite virtually the entire mail delivery system.

The Andrew BE2 Toolkit

The BE2 Toolkit is the heart of the multi-media features of the AMS. Without it, the AMS simply would not have multi-media mail, or would have had to implement the interpretation of a multi-media datastream itself. Thus, clearly, the benefit to the AMS of using BE2 is enormous.

Currently, we know of no user-level drawbacks to the use of BE2 in the AMS. Although the development process was made more painful by the fast-paced evolution of BE2 and the frequent necessity to devote significant time to converting to the latest version of BE2, those problems are basically irrelevant to the AMS as the users see it.

As a programmer's interface, BE2 was in general extremely useful. BE2 is well enough designed to permit the programmer access at whatever level of abstraction he finds necessary. That is, it rarely suffers the deficiency of "protecting" the programmer from the feature he needs. Additionally, it only rarely forces the programmer to use a lower level of abstraction than he desires.

The only major negative factor in BE2 programming is the initial complexity faced by the new programmer. BE2 is a very clever and complex system, with a relatively high "entry cost". It is hard to imagine an uninitiated programmer understanding the most trivial "hello world" BE2 program in less than half an hour. However, this situation can be expected to improve somewhat as better documentation and tutorials become available.

Status Report

Statistics

The Andrew Message System has been in use by the Carnegie-Mellon campus for about a year and a half, and for a lesser time at other sites. In order to convey the size and scope of the CMU installation, we will mention a few statistics about the AMS at CMU as of the fall of 1987.

Currently at CMU there are approximately 1350 bulletin boards on the system; most of these are publicly readable, but about a hundred or so are semi-public. (Semi-public bboards are bboards which are publicly *visible*, but which are only readable by designated users. No statistics are available on the number of completely private bboards, as users may create these completely independently and secretly.)

The 1350 bboards include all of UNIX netnews, all of the Dow Jones information service, and hundreds of Internet mailing lists, in addition to hundreds of CMU-only bboards. Among the CMU-only bboards are a suite of bboards used by the consulting staff to answer users' questions and problems; these bboards are visible to the developers as private bboards, so that the developers can easily intervene with assistance for the trickier questions.

Taken together, annual postings on all of these bboards amounts to at least ten gigabytes, with a new message showing up, on average, about every 40 seconds. Helping users cope with this flood of information are sixteen user-edited "magazine" bboards.

Currently there are about 1800 regular users of the system at CMU. The "typical" user is hard to describe. A large number read only a few "official" bboards, but another large group reads dozens or even

hundreds of bboards. One productive staff member manages to get work done despite reading 377 bboards regularly.

Future Plans

The future of the Andrew Message System holds many new and exciting developments in store. Work is progressing or planned in the following areas:

FLAMES: Work is progressing on FLAMES, the Filtering Language for the Andrew Message System. This will be a powerful and general LISP-like language for operating on the message database. The two primary initial applications will be database queries and automatic classification of incoming messages. In the future we will also be concentrating on improved customization facilities, intelligent selection of messages (as in the Information Lens system [4]), and automatic network-based information services.

Name resolution: Our White Pages technology has proven so successful that we are considering ways to expand it to include names from non-Andrew and non-local sources, leading eventually toward the goal of a "national white pages".

Expansion of Client Base: Efforts are underway to increase the functionality available to AMS users on systems such as the Macintosh and VMS.

Event calendars: Ordinary bboards are inadequate for announcing specific events for future dates; special features are planned to help make event planning simpler and more natural.

Two-dimensional bulletin boards: Eventually, we hope to make our bulletin boards optionally visible in a two-dimensional manner, where responses are clearly associated with the messages to which they respond.

Conferencing systems: It is our hope to eventually bridge the gap between bboard and conferencing systems by providing a way to conduct an interactive conference on a bulletin board.

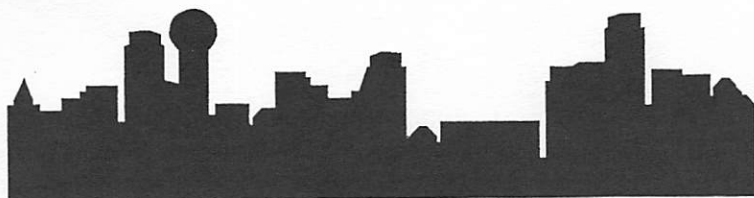
Improved Customization facilities: Although the system is already highly customizable, there is still much room for improvement, particularly in the marking and annotation of messages in the database, and in the user interface to the customization facilities.

Acknowledgments

In addition to the authors of this paper, dozens of other people have contributed to the development of the Andrew Message System. Although space does not permit us to mention them all here, special credit should be given to Brian Arnold, Mark Chance, Bob Glickstein, David Kovar, Sue Pawlowski, Larry Raper, Mike Sclafani, and Aaron Wohl for their work on the AMS. We also wish to acknowledge our indebtedness to the 'advisor' staff (our most demanding and helpful users), the Andrew documentation group, the Vice group, the BE2 group, the ITC management, the networking group, and the support staff, all of whom helped make the AMS a reality.

References

- [1] Morris, et. al, "Andrew: A Distributed Personal Computing Environment", *Communications of the ACM*, Volume 29, Number 3, March, 1986, pp 184 - 201.
- [2] Palay et al., "The Andrew Toolkit: an Overview", *Proceedings of the USENIX Technical Conference*, February, 1988. (this volume)
- [3] Michael Leon Kazar, "Synchronization and Caching Issues in the Andrew File System", *Proceedings of the USENIX Technical Conference*, February, 1988. (this volume)
- [4] Malone, et. al, "Intelligent Information-Sharing Systems", *Communications of the ACM*, Volume 30, Number 5, May, 1987, pp 390-402.
- [5] Rosenberg, et. al, "An Overview of the Andrew Message System", *Proceedings of SIGCOMM '87 Workshop*, Frontiers in Computer Communications Technology, Stowe, Vermont, August, 1987.,
- [6] Lehman, Philip L., and S. Bing Yao, "Efficient Locking for Concurrent Operations on B-Trees", *ACM Transactions on Database Systems*, Volume 6, Number 4, December, 1981, pp 650-670.



An RPC/LWP System for Interconnecting Heterogeneous Systems

Jan Sanislo
Mark S. Squillante†
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

ABSTRACT

This paper describes the design and implementation of the Heterogeneous Remote Procedure Call (*HRPC*) system. *HRPC* has two distinguishing features. First, it allows creation of UNIX applications capable of using several different remote procedure call protocols for network communication. Multiple protocols can be used concurrently within a single program; the fact that multiple protocols are in use is completely transparent to the application programmer. Employing *HRPC* facilitates the "loose" integration of heterogeneous systems using RPC for access to network services.

HRPC's second feature is that it supports the integral use of lightweight processes in a multi-RPC environment. The *HRPC* run-time system uses a "generic" lightweight process interface. This allows *HRPC* applications to employ the specific lightweight process package that is most readily available or most suited to a particular application.

HRPC currently supports two widely used RPC protocols (SUN RPC and Xerox Courier) and has been used to construct or access non-trivial applications. Performance is comparable to that of the supported RPC protocols in the "native" implementations.

Introduction

Over the past several years, one of the major influences on systems development has been the need to provide access to network services in a distributed computing environment. Remote Procedure Call (*RPC*) has become a widely used method for implementing such access [3, 5, 6]. UNIX (or UNIX-based) systems have often been used as the platform on which to build and use *RPC* [12, 9]. *RPC*, with its client-server based model of computation, is especially suitable for achieving what we refer to a *loose integration* of systems. That is, network services are viewed as discrete resources attached to particular computers.

Another major trend has been enhancing UNIX to support lightweight processes (*LWPs*). *LWPs* are threads of control (usually referred to simply as *threads*) that share the same address space and require minimal effort to effect a context switch. Thread implementations have been done both within the UNIX kernel [14] and by using object libraries [7, 8]. Indeed, remote procedure call and lightweight processes are often used in conjunction with each

other. This combination is sufficiently widespread that we will refer to it as an *RPC system*.

RPC systems are the usual mode of accessing network resources in many computing environments. Since most environments include a variety of hardware, *RPC* systems are designed to compensate for machine dependencies such as byte ordering and the number of bits in an "integer". Although *RPC* systems expect hardware heterogeneity, they enforce *systems homogeneity* – the underlying assumption is that all network services employ the *same* *RPC* system. This assumption is exactly the opposite of what occurs in numerous computing installations, where the tendency is toward *systems heterogeneity*.

As a concrete example of such heterogeneity, consider the following: Connected by a local network are a Xerox Dandelion workstation, a SUN workstation running SunOS UNIX, and a VAX mainframe running 4.3BSD UNIX. The SUN and the VAX share a common *RPC* system (SUN *RPC*). The Dandelion and the VAX also share a common *RPC* system (Xerox Courier). What the user wants to do is construct network application (*i.e.*, a distributed set of programs that perform a single logical task and communicate using *RPC*) that can access information on any of these machines.

In this paper we describe our approach to accommodating such system heterogeneity, the Heterogeneous Remote Procedure Call (*HRPC*) system. We

This work was supported by the National Science Foundation under Grants DCR-8420945 and CCR-8611390. Equipment was provided by the Xerox Corporation University Grants Program and by the Digital Equipment Corporation External Research Program.

†Squillante is on a leave of absence as a Member of the Technical Staff, AT&T Bell Laboratories, Murray Hill, NJ.

begin by presenting a short overview RPC, stating the problems encountered in a heterogeneous systems environment, and outlining the HRPC approach to solving these problems. We then focus on details of the design and implementation of HRPC that allow concurrent use of multiple RPC protocols and support for different thread implementations in the HRPC environment. Separate sections cover the performance of HRPC applications built (or made accessible) using HRPC. A final section presents our experiences using HRPC and outlines directions for future work.

RPC Systems and Heterogeneity

Remote Procedure Call (RPC) has become a widely implemented method for accessing network services in a distributed computing environment. The basic idea of RPC is that communication with remote services should have the same syntax and semantics as traditional procedure calls. Simple and elegant in concept, RPC can be difficult to implement. Problems that must be solved include naming and locating network¹ services, ensuring that an RPC is executed only once (or not at all), maintaining the order of sequences of RPCs, compensating for hardware data representation differences between machines, and actually moving data between caller (*client*) and callee (*server*). The mutual agreement between client and server that determines how these problems are resolved is termed an *RPC protocol*. The primitive operations of a particular protocol are implemented by the *RPC run-time* code.

In addition to run-time code, which makes RPC *behave* like a local procedure call, most RPC implementations include a *stub generator* or *stub compiler*. The function of the stub generator is to make RPC *look* like a local procedure call at the syntactic level. Stub generators operate by first reading an *interface description* containing the names and parameters of the remotely callable procedures associated with a particular application. From this information, the stub generator produces source code modules, called *stubs*. The stubs hide the details of the RPC run-time from the application programmer. Each remote procedure has an associated stub routine; an RPC is started by performing a local procedure call from the application to the stub routine. A client stub routine contains a sequence of calls to RPC run-time primitives that (typically) establish connections to services, marshal parameters into a call message, send the call message, wait for a reply and de-marshal result parameters. Similar stubs are used by a server program to wait for a call message, de-marshal input parameters, execute the actual operation requested, marshal return parameters and send a reply.

It is worth noting the sequence of operations

performed within the client and server stubs. They reveal the flow of control required by RPC semantics. RPC is a synchronous process² with a client suspending execution to wait for a reply and a server executing single requests sequentially. This can have an adverse impact on the performance, CPU utilization and programming structure of RPC applications.

- The latency of network communication combined with the inherent delay of processing a request means that a client will spend substantial amounts of time being blocked from execution.
- Within a server, it may be necessary to perform operations (such as RPCs to other servers) that block its execution. The server can become a bottleneck if many clients access it.
- Some applications are most naturally structured as a set of independent parallel activities rather than a fixed sequence of dependent actions.

These factors lead to interest in using lightweight processes to allow multiple threads of control within "heavyweight" UNIX-level processes representing client and server. In such systems, each remote procedure call is managed as a separate thread. It is this thread that is blocked when necessary, rather than the entire client or server process.

Although RPC systems have proven useful, they suffer from a common problem. Each implementation assumes a *homogeneous* RPC environment. New combinations of hardware/operating system cannot be integrated into the existing environment until they implement the "standard" RPC system. Reasons for the insular nature of these existing RPC systems are not difficult to find. RPC technology is relatively young. Attempting to satisfy diverse research interests and having no compatibility requirements, everyone has invented their own RPC protocol. While there is no denying the knowledge gained from these separate efforts, the mutual incompatibility of different RPC systems leads to major complications in real-life situations.

"Heterogeneity Through Homogeneity"

In several kinds of computing installations (*e.g.*, a university computer science department), it is inevitable that the hardware and operating system environment will be *heterogeneous*. For research or economic reasons, different types of equipment will be acquired. Sometimes the equipment is purchased because of its singular properties, such as special display hardware, vector processors, or ability to run unique software. On other occasions equipment may be acquired to increase capacity, such as printers, tape drives, disk space or computing cycles. In either case the challenge is the same – to make these properties available to other systems in a consistent and easily used manner. We refer to this as *loose integration*, since sharing is done at the level of certain discrete

¹We consider a network to be a logical rather than physical entity. There is no requirement that communication takes place "over a wire". Some RPC systems use shared memory as the communication medium.

²This synchronosity is one of the keys distinguishing RPC from more general "message passing" communication protocols.

resources within each system.

RPC, with its client-server model, is a particularly suitable mechanism for achieving such loose integration. Indeed, most (but not all) systems already have some sort of RPC capability, including well-defined interfaces to their interesting hardware and software resources. The problem is that each RPC system is mutually incompatible with the others at the protocol level. Incompatibility is usually attacked by porting RPC implementations from one system to another. This approach can require substantial effort. RPC systems consist of large amounts of software (stub generator, run-time code, thread support, administrative utilities, etc.), much of which is highly dependent on a particular operating system interface. Once the RPC system is ported, more work is needed to move applications to the new environment.

The task of porting an RPC system is made more difficult by the fact that logically independent functions of the system become tightly coupled, usually for performance reasons: Stubs are generated that depend on particular RPC primitives and structures. Parameter marshaling depends on the fact that one data representation format is used. The semantics of a particular network protocol are assumed. Interactions between RPC protocol and LWP support become intertwined and mutually dependent. Thus, porting can require changing or implementing large amounts of code even if functionally equivalent software (e.g., an thread package or network transport protocol) already exists on the target system.

The HRPC Approach

HRPC takes a different approach to the problem of providing RPC access within a cluster of heterogeneous systems. Recognizing that there are circumstances under which it will be extremely difficult or impossible to port code, HRPC seeks to *accommodate* heterogeneity. Accommodating heterogeneity means that we do not attempt to enforce a "standard" RPC system. Rather, we wish to allow the simultaneous use of multiple RPC protocols in a way that is transparent to both the programmer and to existing RPC-based applications. RPC protocol transparency has several benefits. First, HRPC programmers are presented with a single, consistent interface to underlying RPC systems. Second, existing RPC-based applications need not be modified in order to access or be accessed by programs using HRPC. Third, new applications can be created using the native programming environment and tools of the system on which that application will run.

HRPC provides protocol transparency by structuring itself as a set of "generic" RPC run-time primitives. The actual implementation of each primitive is not decided until the HRPC application is run and the choice is based on the protocols client and server have in common. Code that implements a particular RPC protocol is then "plugged in" to the generic primitives. Code for each specific protocol is contained in a

library created either from scratch or by placing a thin veneer over an existing set of RPC run-time primitives.

A similar approach is employed in accommodating existing LWP packages. By carefully limiting the interactions required between a threads implementation and the rest of HRPC and making minimal demands on the semantics of thread primitives, HRPC is capable of using an existing LWP package without modification to either HRPC or the LWP package used. There is however a subtle distinction between how RPC protocols and LWP packages are accommodated. While *multiple* RPC protocols will routinely be used simultaneously within a single instantiation of an HRPC program, only a *single* LWP package can be used.

Accommodating RPC Heterogeneity

The basis of the HRPC system is the idea that *any* RPC protocol can be decomposed into three functional components. Each component is logically independent of the others:

1. *Control Component.* This component defines the structure and semantics of messages passed between client and server during a remote procedure call. For instance, the control component defines such things as how "call" messages are distinguished from "reply" messages, message sequence identifiers, retransmission strategies at the RPC message level (rather than network packet level).
2. *Data Representation Component.* This component, often referred to as *over-the-wire* or *OTW* representation, determines the conventions used to ensure data compatibility between client and server. At the lowest level, these conventions isolate machine dependencies by defining, e.g., that integer parameters are represented high-byte first within a RPC message. However, the OTW component also compensates for *programming language* dependencies by defining the representation of structured data such as arrays or records within an RPC message.
3. *Transport Component.* The transport component represents the mechanism by which messages are transmitted over the network. Transport is usually represented by a network transmission protocol such as UDP or TCP, although it could just as well be a set of primitives that operate on shared memory.

In traditional RPC facilities, all decisions regarding the implementation of the various components are made at the time the RPC facility is designed — choices such as whether a stream-oriented or datagram-oriented transport protocol will be used, byte ordering of data, where control information will appear within messages, etc. Making these choices early has the disadvantage of inevitably leading to the kind of "monolithic" implementation that is difficult

to modify or port to a new operating system environment. However, it has the advantage of reducing and simplifying the amount of RPC protocol overhead when the program is actually run. For instance, in such systems the only information needed by a client to access a server is the *network location* (including communications end-point) of that server; no other decisions concerning details of communication between client and server need be made. The process of acquiring this location information is referred to as *binding*. The result of binding to a server is a data structure, called a *Binding*, containing information describing the logical connection to a server.³ Bindings are typically "held" by the client, and are passed to the stub as an explicit parameter of each remote procedure call.

By increasing the amount of work done at bind-time, significant flexibility can be achieved. An example is the DEC SRC RPC system. The designers of this facility realized the advantages of being able to use any one of a number of transport protocols. The SRC system delays the choice of transport protocol (e.g., shared memory, packet exchange, TCP) until bind-time, when the choice can be made based on availability and performance. The actual choice of protocol is invisible to both the client and the server.⁴ In effect, the SRC system has *factored out* any knowledge of a specific transport protocol.

The HRPC system extends this factorization to the other RPC functional components. The choices of transport protocol, data representation, and control protocol are delayed until bind-time, allowing a single HRPC program to communicate with a wide variety of "traditional" RPC programs.

The basis of the HRPC factorization is a single abstract model of how *any* RPC facility works, as expressed in the three previously noted functional components. To isolate the actual implementation of each of these components from the others, we began by defining a procedural interface to each component, which is given in Table 1 and elaborated upon in the paragraphs that follow. At first glance, these interfaces may appear overly specified in relation to more traditional RPC facilities. This is because such systems tend to keep interfaces "small" by building a great deal of knowledge concerning remote procedure calls (such as the structure of control headers) into the client and server stubs at compile-time. HRPC requires more extensive and detailed interfaces since such knowledge is not available until later.

The Control Component.

The control component has three routines associated with each *direction* (send or receive) and *role* (client or server) within a remote procedure call. (The

latter distinction is needed since the control information in request and reply messages is generally different.) For example, the *Call* routines are used by the client side when sending the initial call message to some service. *InitCall* performs any initialization necessary prior to beginning the call. *CallPacket* performs any functions peculiar to the specific protocol when it is necessary to actually send a segment of the complete message. And *Finish-*

Control Component Interface		
<i>InitCall</i>	<i>CallPacket</i>	<i>FinishCall</i>
<i>InitAnswer</i>	<i>AnswerPacket</i>	<i>FinishAnswer</i>
<i>InitRequest</i>	<i>RequestPacket</i>	<i>FinishRequest</i>
<i>InitReply</i>	<i>ReplyPacket</i>	<i>FinishReply</i>
<i>GetPacket</i>	<i>PutPacket</i>	<i>CloseRpc</i>

Call terminates the (outgoing) call. Similarly, *Request* routines perform functions associated with a service receiving a call, *Reply* routines are used when the server replies to the client request, and *Answer* routines are concerned with receiving that reply on the client end. *CloseRpc* is used by user-level routines to notify the RPC facility that its services are no longer required. It is important to note that these routines are more than just "packet pushers". They are responsible for implementing the peculiarities of individual control protocols, such as probe packets, reply timeouts, and error detection. For reasons to be made clear later, the *GetPacket* and *PutPacket* serve as aliases for one of *CallPacket/ReplyPacket* or *RequestPacket/AnswerPacket* respectively. All routines take a single input parameter, a pointer to an HRPC Binding.

The Data Representation Component.

OTW Component Interface		
Cardinal	LongCardinal	Integer
Boolean	Unspecified	LongUnspecified
Array	Choice	Enumeration
Record	NilRecord	Procedure
Deallocate	CloseOtw	Error
LongInteger	String	Sequence

The number and purpose of the data representation procedures are driven by the interface description language (IDL). There is one routine for each primitive type and/or type constructor defined by the IDL.⁵ The parameters to each routine consist of a pointer to an HRPC Binding and the address of data object of the specified type. Each routine translates an item of a specific data type in host machine representation to/from some standard over-the-wire format. Each routine is capable of encoding or decoding data in the manner of Xerox Notes objects [16] or SUN XDR routines [13]. Whether encoding or decoding is being done is determined by the state of the call.

⁵HRPC uses modified Courier [15] as the IDL. The routine names given here reflect this choice.

³To avoid confusion between the binding process and the resulting binding data structure, we refer to the latter as a *Binding*.

⁴In contrast to e.g., SUN RPC, where the choice of transport protocol is explicitly made at the application level.

The Transport Component.

Transport Component Interface		
OpenLink	CloseLink	
InitSend	SendPacket	FinishSend
InitRecv	RecvPacket	FinishRecv
BufAlloc	BufDealloc	MaxBufferSize

The functions of the transport routines are immediately obvious from their names – opening or closing a logical “link” according to the peculiarities of a particular transport protocol, and sending or receiving a packet. Each of these routines takes a single input parameter, a pointer to an HRPC Binding. The *Init* and *Finish* routines are called to initialize or terminate transmissions/receptions related to a single call message. This allows transport-specific processing to occur in an orderly manner, e.g., setting the *EndOfMessage* bit in the last packet of a message using the Xerox Sequenced Packet Protocol. The memory buffers used by messages are considered to be the property of the transport level and are acquired or released via *BufAlloc* or *BufDealloc* calls.

Interaction Among RPC Functional Components

The reason for these clearly defined procedural interfaces is that we require an HRPC stub and any combination of control protocol, data representation, and transport protocol components to be able to function together. The interaction between these entities

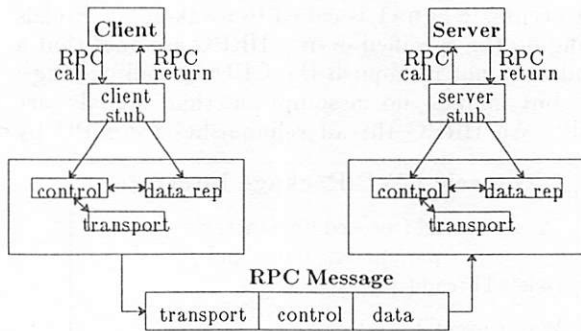


Fig. 1. Interaction among RPC functional components in HRPC.

is depicted in Figure 1, where the direction of the arrows indicates the direction of calls during the “call” portion of an RPC (as opposed to the “return” portion). Note that there is a direct correspondence between the logical segments of an RPC message (which may consist of multiple transport messages) and the three lower-level functional components of the HRPC system.

The transport component, as mentioned above, provides network buffers (including an indication of their maximum size) and sends and receives these buffers. The control component is responsible for calling transport component routines that initialize for sending or receiving, and for obtaining appropriate buffers. The control component then makes use of the

data representation component to insert protocol-specific control information into the message being constructed. The data representation component merely fills or empties data buffers – it is unaware of the distinction between user data and RPC control information⁶. Note that the data representation routines never directly access transport functions. When a buffer is full, a control-level operation is called to dispose of the buffer. A control operation must be called because the data representation routines are not expected to know details concerning the placement of control information within a message.⁷ Similarly, the data representation routines are ignorant of whether they are operating in a client or server context; this distinction may be important to the control routines. The data representation component communicates with control component via the alias routines *GetPacket* (actually implemented by *CallPacket* or *ReplyPacket*) or *PutPacket* (actually implemented by *AnswerPacket* or *RequestPacket*). These generic routines isolate the buffer-filling routines from the *role* context that they are operating in, although of course they must be aware of the *direction*, sending or receiving.

Note that there is a situation in which control and data representation components can call each other in a mutually recursive manner. It is possible that a user data item will be split across packet boundaries and that the packet containing the completion of the data will be prefixed by control information. In this case, the control routine called when the original packet is exhausted will (recursively) use data representation routines to strip off this control header.

The Structure of HRPC Bindings

The Binding used by the HRPC facility must be considerably richer in information than the Binding used by more traditional RPC facilities: since the particular choices of each of the three components are delayed until bind-time, an HRPC Binding must include not only the *where* of a remote procedure call but also the *how*.

An HRPC Binding contains three separate blocks of procedure pointers, one for each of the functional components. Calls to the component routines are made indirectly via these procedure blocks. It is this indirection that allows the actual implementation of the component routines to be selected at bind-time. Also associated with each component in the Binding is a private data area. A component uses this area to hold information specific to the implementation of that component, e.g., the data type representing the transaction identifier for an RPC protocol, or a transport-dependent network address.

Bindings are opaque at the user level.

⁶Control information is assumed to use the same data representation as user data.

⁷For example, perhaps the control protocol uses “trailers” rather than “headers”. The fact that “trailer” space may have to be reserved is one of the reasons why the control component is responsible for obtaining and perhaps manipulating buffer space.

Application programs never directly access the component structures of a Binding – they deal with Bindings only as atomic types, and acquire and discard them via HRPC system calls. Once a user has acquired a Binding, it is used to make an HRPC call by supplying it as an explicit parameter to a *stub routine* specific to a remote interface.

The Structure of HRPC Stubs

Stub routines function as translators between the user's view of remote procedure calls as regular procedure calls and the actual implementation of a remote procedure call as an exchange of messages containing serialized data. *Client stubs* implement the calling side of an HRPC. *Server stubs* implement the called side of an HRPC. The HRPC stub generator (based on one developed at Cornell University [6]) uses the Courier [15] as the interface description language. Stubs have built into them detailed knowledge of Binding structures and of the interface to the underlying HRPC components. However, the key feature of HRPC stubs is that they are completely de-coupled from the details of specific RPC protocols. For a given remote procedure, the client stub routine is always the same: A initial call to the control component `InitCall` routine in the Binding, followed by a sequence of calls to OTW component routines to marshal input parameters, ending with `FinishCall`, which terminates the call message. The client stub then waits for an answer by calling `InitAnswer`, demarshals result parameters, calls `FinishAnswer` and then returns to application level code. An analogous series of component routine calls is performed within a server stub.

The de-coupling of stub from RPC protocol means that the same stub routine can be used for RPCs involving totally different protocols. As an extreme example of this independence, that it is possible to dynamically load an RPC component suite into a running program and make remote calls using stubs that had been compiled and linked before the component suite was created.

Lightweight Processes in the HRPC System

There are two facets to the treatment of lightweight processes in the HRPC system. The first is that, just as it is possible to use multiple RPC protocols, we wish to be able to transparently use different packages implementing lightweight processes. It is important to note that "transparency" with regard to threads refers not to the application program as it does with RPC, but rather to the HRPC system itself; the HRPC run-time must be capable of functioning with an arbitrary LWP package. Since the application programmer will have direct access to the underlying LWP primitives, this requires an interface between the programmer and the HRPC system. The second and perhaps more interesting facet of lightweight processes within the HRPC system is how threads are used within the context of multiple RPC protocols. This interaction leads to a subsystem of

HRPC that manages lightweight processes and provides an interface to LWPs used by the RPC protocol subsystem.

Accommodating LWP Heterogeneity

Our approach to accommodating various LWP packages is essentially identical to accommodating multiple RPC protocols. Any LWP implementation can be thought of as providing a set of abstract operations that create, schedule and terminate threads of control. The application programmer provides the HRPC thread management subsystem with a set of routines that implement these abstract operations. Thread management uses these routines both to create threads that are internal to HRPC and to manage user-created threads engaging in remote procedure calls.

The "generic" interface presented by the external LWP package to the HRPC thread subsystem consists of the seven procedures shown in Table 1. Threads are created and deleted within HRPC via the `CreateThread` and `DeleteThread` routines. The lightweight process created by `CreateThread` has the specified priority, stack size and name (if these attributes are supported by the LWP package being used), runs the specified procedure with the arguments pointed to by "parameters", and is uniquely identified by the returned pid. The LWP subsystem makes no assumptions about which thread executes after calling `CreateThread`. `Wait` is called within HRPC to block the running thread until the specified event occurs. `Signal` is called to awaken all threads waiting on the specified event. HRPC assumes that a thread does not relinquish the CPU by calling `Signal`, but makes no assumption that signals are "held". An HRPC thread relinquishes the CPU by

Generic LWP Package Interface	
CreateThread (procedure, stacksize, priority, parameters, name, pid)
DeleteThread (pid)
Wait (event)
Signal (event)
Scheduler ()
BlockScheduler ()
UnblockScheduler ()

Table 1. HRPC LWP Package procedural interface.

calling `Scheduler`.

The first five procedures sufficiently define the interface to a non-preemptive, priority-based external LWP package. To accommodate preemptive packages requires the two additional procedures `BlockScheduler` and `UnblockScheduler`. HRPC requires exclusive access to internal thread-related data structures in order to prevent possible corruption. Since a preemptive scheduler may be invoked asynchronously, routines accessing these structures

call `BlockScheduler` to prevent context switches. When exclusive access is no longer needed `UnblockScheduler` is called. The thread subsystem ensures that a thread does not relinquish the CPU between calls to `BlockScheduler` and `UnblockScheduler`. We assume that preemptive LWP packages either define such exclusive access routines or provide primitives that can be used to obtain the necessary functionality. If a non-preemptive package is being used, the user would specify `BlockScheduler` and `UnblockScheduler` routines as null routines.

HRPC maintains a separate block of procedure pointers to the routines implementing the interface to the external LWP package. These pointers are not part of an HRPC Binding because *all* HRPC threads share these routines. Calls to the external LWP package are made indirectly via the procedure block, allowing the actual implementation of the component routines to be specified by the application programmer during program initialization by calling `SetupLwpSupport`, providing as input the addresses of seven routines implementing the generic interface along with other information describing the traits of the particular LWP package. If `SetupLwpSupport` is not called, HRPC operates in what we call *single-thread mode*.

This clearly defined interface facilitates the functioning as a single unit of three separate components with minimal knowledge of each other – HRPC protocol support, HRPC thread management and the external LWP package. The interaction between these entities is depicted in Figure 2, where the direction of the arrows indicates the direction of calls between components. The following sections provide details of this interaction.

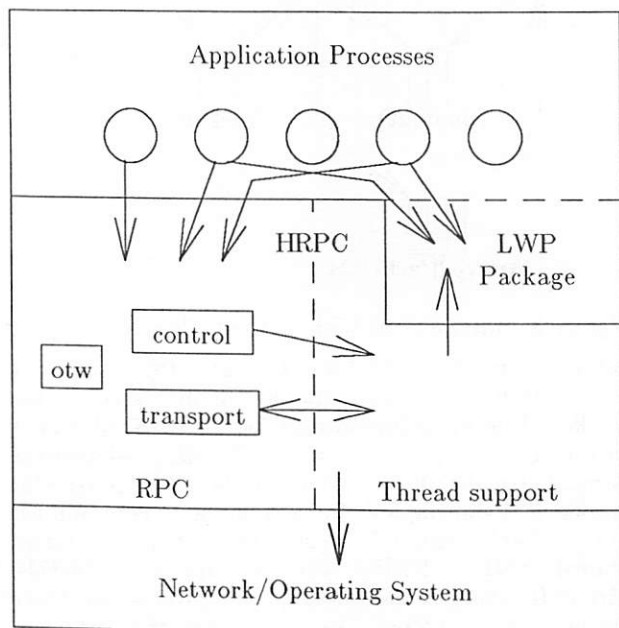


Figure 2: Interaction in HPRC Applications

Thread Management Within The HRPC System

As mentioned previously, the user is expected to create threads that act as RPC clients and servers. The HRPC system is responsible for scheduling these threads within the context of remote procedure calls. Scheduling is required whenever a thread must perform a blocking network input operation such as `select(2)`, `read(2)` or `recv(2)`. A thread making a direct call to one of these operations risks blocking the entire UNIX process until input arrives. To prevent this situation, RPC protocol routines do not call input primitives directly. Instead, calls are made to one of two routines, `Select` or `SelectAndReadPacket`, defined by the HRPC thread management component.

The HRPC `Select` routine is roughly equivalent to the UNIX `select(2)` kernel call. The main difference is that if no input is available, only the calling thread is suspended; the `Wait` routine is used to block the thread. `Select` is used almost exclusively by server threads when waiting for a request. The term "request" here refers to one of two things: a logically complete RPC message (rather than a single network packet that is part of the message) or, if the underlying transport protocol is connection-oriented, a transport-level event that signifies connection establishment but conveys no RPC-level data. Either event will unblock the thread. However, HRPC thread management cannot distinguish the event type. The RPC protocol routines must determine this.

The `SelectAndReadPacket` routine is used when an RPC thread must wait for the next in a sequence of transport-level packets to arrive. As with `Select`, the calling thread is blocked until input is available (or a timeout interval expires). When a packet arrives it is read from the network socket and passed to the thread. The specific kernel-level input primitive needed to retrieve the packet is actually a function of the transport protocol being used in the remote procedure call. Thus the LWP subsystem does not directly perform the input, but instead "calls back" to a transport-specific `GetPacket` routine contained in the HRPC Binding being used by the calling thread.

Internal Structure of HRPC Thread Support

Upon initialization of HRPC thread support, `CreateThread` is used to create an extra thread, called `SelectManager`, that handles timeouts and blocks the UNIX process. `SelectManager` runs at a lower priority than all other threads in the system and thus is awakened only when all other threads are blocked. When `SelectManager` is awakened, it first checks to see if the timeout interval of any blocked thread has expired. To facilitate this, an ordered *TimerList* mechanism is used by the `Select` routines and `SelectManager`. If a timeout has expired, the `SelectManager` uses `Signal` to awaken all such threads. Otherwise, the `SelectManager` coalesces all `Select` requests together and performs a single

UNIX `select(2)`. When the `select(2)` returns, all threads affected by the result are awakened. In either case, the SelectManager then calls Scheduler to relinquish the CPU. When SelectManager is awakened again, the entire process is repeated.

If the external LWP package being used does not have a priority-based scheduler, SelectManager must determine if it is the only ready thread before executing `select(2)`. This is necessary to prevent blocking the UNIX process when some other thread is able to execute. To check for runnable threads, SelectManager simply notes whether the Scheduler call returned immediately. An argument to `SetupLwpSupport` specifies whether the package being used supports priority-based scheduling.

The concurrency introduced by multiple threads can increase the number of packets destined for the single UNIX process. This may cause packet loss due to the overflow of kernel buffers allocated to network sockets created for RPC purposes. It is important that the design of HRPC thread support minimize packet loss. One possibility is to awaken threads, either threads dedicated to reading incoming packets or threads to which the packets are destined, when packets arrive. This, however, may be insufficient since packets remain in kernel buffers until awakened threads acquire the CPU and it is possible that this will not occur before the socket buffers fill. We chose to copy incoming packets into the address space of the UNIX process immediately upon arrival, thus freeing socket buffer space. To do this, thread support uses the asynchronous I/O facilities of UNIX and associates a special SignalHandler (SH) with the arrival of packets. SH's primary task is to copy incoming packets into the UNIX process address space.

Since the SH is invoked asynchronously and accesses internal data structures as described below, there must be a mutual exclusion mechanism to prevent corruption if other threads of control are also accessing these structures. One possibility is to use `sigblock(2)` to prevent SH from running, but this incurs the overhead of a UNIX kernel call. Instead, we use a more efficient technique. Whenever a thread attempts to enter a critical section, it calls a routine that atomically sets a "hold signals" flag. When SH runs, it first checks to see if the "hold signals" flag is set. If so, SH sets "signal pending" flag and returns without taking further action. Whenever a thread exits a critical section, it invokes a routine that checks the "signal pending" flag and calls SH if the flag is set. When SH returns, the routine resets the flag and returns. This scheme allows us to obtain exclusive access to various system data structures, while saving the cost of making a UNIX kernel call.

Routing packets to appropriate threads within a reasonable amount of time is somewhat complex since threads using different RPC protocols may be active simultaneously. This raises the question of who should awaken threads waiting for packets that have arrived. One possibility is to make this a task of the

SelectManager. Since it is possible that SelectManager will remain blocked for a significant period of time due to its low priority, awakening blocked threads whose packets have arrived may be delayed. If the delay is long enough, remote processes may retransmit their packets since the threads they are attempting to contact have not responded. Hence, the SelectManager does not make a good packet router. Another possibility is to have the SH determine for which thread a packet is destined, pass the packet to this thread, and awaken it. However, this requires that SH know specifics (*e.g.*, message format, OTW representation) of the RPC protocol associated with a particular socket. We refer to these details collectively as the socket's *speaktype*. Instead of placing *speaktype* knowledge in the SignalHandler, we distribute the routing task across a set of special threads, called *speaktype managers* (STMs). `CreateThread` is used by the HRPC system to create an STM thread when the first socket of a particular *speaktype* is opened. Similarly, `DeleteThread` is used to eliminate an STM thread when the last corresponding socket is closed.

Since HRPC thread support itself employs threads, there must be some way for these threads (SH, a set of STMs and a SelectManager) to communicate with each other and the external LWP package. The SH needs to pass packet information to the appropriate STM after it has copied an incoming packet into its address space. Similarly, an STM

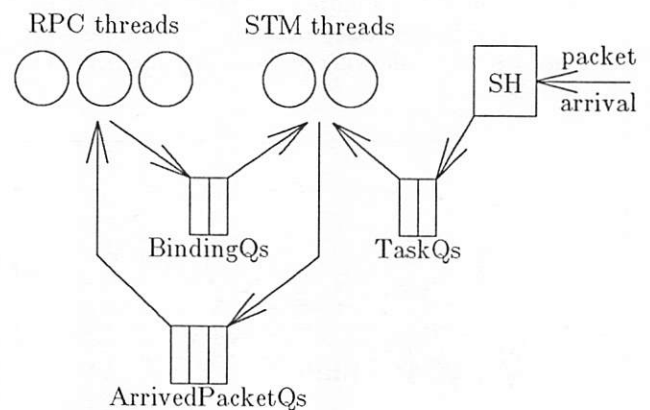


Figure 3: Interactions within HRPC Thread Support

needs to pass packet information to a thread after it has determined which thread should receive the packet. Finally, a thread must notify its STM that it is waiting on a packet and in doing so, must provide some distinguishing information about the particular packet it is waiting for. Intra-component communication is implemented by three types of queues maintained within HRPC thread support: *TaskQs*, *ArrivedPacketQs* and *BindingQs*. Access to these queues is protected, in part, by the `BlockScheduler` and `UnblockScheduler` routines. Figure 3 illustrates the major components of HRPC

thread support, including the various queues; the direction of the arrows indicates the flow of data.

When the SH is invoked, it first determines which sockets have received packets. Associated with each open socket is the speaktype of the lightweight processes using the socket and associated with each speaktype is a corresponding "read" procedure. This procedure is essentially the `GetPacket` routine from an appropriate HRPC Binding and is used to isolate the SH from knowing how to read packets for each of the speaktypes. For each ready socket, the SH uses the corresponding read procedure to copy the packets stored on the socket into buffers of the appropriate size; this is the only time the packet is actually copied. The SH then adds packet information (e.g., packet size and pointer to packet buffer) for each of these packets onto the TaskQ of the STM associated with the socket's speaktype. The SH *causes* this STM to be awakened and returns after processing all ready sockets in this manner. The SH does not invoke the LWP package's `Signal` primitive because that would allow asynchronous access to the package's internal data structures, making them a critical section. Instead, the SH notes that an STM should be awakened by setting a corresponding bit in a global bit-mask. When a thread starts executing within the LWP subsystem, this mask is checked and all STMs whose bits are set are awakened via `Signal`.

When a thread must wait for packet arrival, it places information about itself on the BindingQ associated with its STM. This includes constructing a table of tuples, *IdTable*, used to uniquely identify the packet destined for this thread. The first component of each tuple, *GetPacketIdValue*, is a procedure that is called to obtain an identifying datum from the packet. The second component, *IdValue*, is a value to be matched against the datum obtained from *GetPacketIdValue*. If the value returned by *GetPacketIdValue* does not match the corresponding *IdValue*, the packet cannot be for the thread associated with *IdTable*. This scheme, along with the information maintained in associated HRPC Bindings, provides STMs with a flexible, general purpose mechanism for determining which thread should receive an incoming packet. More importantly it isolates the STM from knowing how RPC threads will uniquely identify their packets and appropriately places this responsibility on the RPC protocols. After adding its *IdTable* to the correct BindingQ, the thread blocks itself by calling `SelectAndReadPacket`.

When an STM acquires the CPU, it first accesses its TaskQ to obtain information about an arrived packet. It then searches through its BindingQ (ordered on nonincreasing table size) looking for the first element whose *IdTable* matches the packet being considered. No element will be found if, for instance, all server threads are busy, and the packet information is left on the TaskQ for Otherwise, the STM determines the thread corresponding to the matching element on its BindingQ and transfers the packet

information from its TaskQ to the thread's ArrivedPacketQ by copying pointers to the data. The STM then uses `Signal` to awaken the thread. The process is repeated for the remaining packets on its TaskQ. After the STM has walked through its TaskQ, it calls `Wait` to block until awakened by SH or by the SelectManager when a timeout expires. The latter can occur when an STM leaves an unprocessed task on its TaskQ, since the STM cannot *rely* on the SH to awaken it in the future.

When the thread awakened by the STM begins executing, it accesses the packet on its ArrivedPacketQ. ArrivedPacketQs allow the STM to pass multiple packets to a single lightweight process, even before the thread is able to use them. When such an RPC thread attempts to wait on the arrival of a packet, the `Select` and `SelectAndReadPacket` routines will not block the thread.

In addition to the three types of queues, the LWP subsystem maintains *BufferPools* to efficiently manage the increased number of packet buffers induced by the structure of our approach. A BufferPool is associated with each active speaktype and contains a list of available packet buffers. All packet buffers on a particular BufferPool are a fixed size defined by the `MaxBufferSize` of the associated speaktype. When the SH attempts to read from a ready socket, it first obtains a buffer from the BufferPool associated with that socket's speaktype. If no buffers are available, the BufferPool mechanism creates a buffer of the proper size by calling the appropriate `BufAlloc` procedure. All routines needing to allocate or deallocate a packet buffers do so through the BufferPool. The BufferPool mechanism prevents the proliferation of buffers by dynamically adjusting the number of free buffers according to the state of the system.

An Example HRPC Application

To illustrate how the capabilities of HRPC can be used in structuring programs, we describe an example HRPC program. The emphasis here is not so much on *what* is done but rather on *how* it is done. The example focuses on several features of the HRPC system:

- Transparent use of multiple RPC protocols.
- Use of threads to easily implement both client and server operations within a single UNIX process.
- Ability of HRPC-based servers to function as "bridges" between system types that do not share a common network transport protocol.

The example application implements a facility for obtaining Ethernet usage statistics from several different types of machines. The application consists of three different programs that communicate using HRPC:

- *Statistics Server*. An instance of this server runs on every machine from which statistics will be collected. When called it probes operating system-

dependent data structures containing Ethernet information and returns that information as results.

- *Client Program.* This is the program actually run by the user when he/she wishes to view Ethernet statistics. The user specifies a list of machine names from which statistics are desired. This list of names comprises the RPC input parameters. The statistics for each machine are returned as call results and displayed. The user program does not make direct remote procedure calls to statistics servers. Instead it calls an intermediary client agent.
- *Client Agent.* The client agent runs on some subset of machines in the network. Inside the client agent are multiple threads of control, each of which can be thought of as a separate server waiting for incoming remote procedure calls from user programs. When a call is received, an idle thread is activated to service the call, the input parameters of which consist of a list of machine names. The activated server thread traverses the list of machines. For each machine a new thread is created. The new thread performs a "GetStatistics" call to the statistics server at a given machine. In this way all calls to statistics servers proceed in parallel. Using the synchronization primitives of the external LWP package the original server thread within the user agent waits for all statistics calls to complete. The results of each statistics call are then composed into a reply to the original call from client program to client agent.

At first glance it may seem unusual to have three programs involved in an application where two (user program and statistics server) would normally be used. However, the user agent performs an extremely useful function that is not apparent until details of the network environment are examined. On our local network we have (among others) three types of systems: Xerox Dandelion workstations using Courier RPC with XNS transport protocols; SUN workstations running SunOS 3.3 using SUN RPC with IP transport; VAXen running 4.3BSD with both XNS and IP transport, and implementing HRPC which is capable of emulating both SUN RPC and Courier RPC. By running user agents only on the VAXen, it is easy to provide the SUN systems with access to data that resides on the Xerox systems - data which SUN users would be unable to obtain without implementing the Courier RPC protocol (which, for all practical purposes, includes the XNS protocol suite) on SUN workstations.

It is important to note that this flexibility does not require any special programming. Because the HRPC system provides a programming environment that makes it possible to ignore specific RPC protocols, application code is easily portable between UNIX machines. In our example, the user agent and client program C code will compile and run without change on a SUN. The statistics server would require

changes, but *only* to the machine-dependent parts that retrieve Ethernet numbers from kernel data structures. The only surprise would be that the SUN version of the user agent would generate a run-time error if asked to communicate with a Xerox-based statistics server. If the XNS protocol suite were suddenly available on SUN systems then the entire application could be transferred without change.

Experiences Using HRPC

HRPC has been in use for almost two years. During that time it has been used to create a variety of non-trivial applications. These include:

- A distributed Heterogeneous Name Server (HNS) [10]. By using HRPC, the HNS is capable of interrogating existing name servers such as the Berkeley BIND name service of the Xerox Clearinghouse.
- A distributed Heterogeneous Mail Service (HMS) [11]. By using HRPC as its local communication mechanism, the HMS is able to integrate existing local mail services such as the UNIX and Xerox mail systems.
- A remote execution facility called THERE [1]. THERE has been used in programming classes to provide access to an Ada compiler that has only a single-machine license.
- A prototype file system. The file system uses the "mix and match" flexibility of the RPC component interface to store disk data in a machine-dependent format while providing access to that data in a machine-independent format.

Numerous smaller applications have also been created, such as the network statistics server previously mentioned and a "face finger" utility that uses the prototype file system to store bitmap images. Since HRPC has been available essentially no applications have been developed using the "native" RPC facilities (SUN RPC and the Cornell University UNIX Courier implementation) available on our UNIX systems.

One reason for the local acceptance of HRPC is that, in addition to flexibility and consistency, it offers performance comparable to "native" RPC systems. Our benchmark programs⁸ show that HRPC is using no LWP support is in the worst case 3% slower than native implementations of these benchmarks and, in some cases is actually faster. When the HRPC benchmarks use one LWP thread, the overhead increases by approximately 3.73 milliseconds per call, where the average cost of a simple, parameterless RPC is 25ms. These figures indicate that the design of the HRPC system does not pay a prohibitive cost for the adaptability that it provides.

The HRPC system itself has been extended to include a "raw" RPC protocol that can be used to communicate with some BSD-style network utilities that were not originally constructed using a real RPC.

⁸One a simple call and return with no parameters and the other transferring 37,000 bytes of data.

The framework provided by HRPC has been used to bring up SUN RPC on a Tektronix Pegasus workstation and an interface between Smalltalk and HRPC has been constructed. The HRPC stub generator has been extended to produce stubs coded in Franz Lisp. More information on how HRPC has been employed can be found in [3].

Although the HRPC system has had many successes, it also has some shortcomings. Perhaps the most irritating is the "least common denominator" syndrome that results from trying to make the peculiarities of the different RPC protocols transparent to the user. A specific instance of this occurs when handling application level errors at run-time. The Xerox Courier RPC protocol provides a specific mechanism by which applications can return either the normal call results or an error-dependent set of parameters if there is some untoward occurrence detected by the server. The SUN RPC system does not provide such an "alternate return". Thus, the capabilities of the Courier protocol cannot be utilized since program behavior would be dependent on the RPC protocol in use at the time of the call. Another area where the "LCD" approach can cause problems regards the use of external LWP packages. Out general, seven-procedure interface will prevent HRPC thread support from directly using special, perhaps more efficient features of a particular package. This does not, of course, preclude the application programmer from using them.

We previously mentioned that the SelectManager is meant to run at the lowest possible priority. We also described how it functions when the LWP package being used does not support priority scheduling. Our solution "side steps" the problem but forces the SelectManager to obtain timing information before and after its call to Scheduler. This is awkward at best and increases the overhead incurred by the SelectManager. Another potential problem is that thread support is biased towards a Signal primitive that has "no-yield" semantics. For instance, each blocked STM that has received a new task is awakened by a separate Signal call. This will be much more efficient if Scheduler is not invoked as a side effect. HRPC will function correctly in the latter case, but the number of LWP context switches will be increased.

Future Work

Our department will soon take delivery of a group of Fireflies, experimental prototype multiprocessor workstations developed by DEC's Systems Research Center. These machines have little in the way of normal peripheral equipment such as tape drives and printers. However, their normal mode of communication is the DEC SRC remote procedure call system cited previously. We anticipate adding the SRC protocol to the HRPC suite, allowing the Fireflies to access existing resources within the department.

The advent of Fireflies (along with an already acquired Sequent multiprocessor) also presents challenges to HRPC support for lightweight processes. The current design of the LWP subsystem assumes execution on a uniprocessor. Access to critical sections is protected by preventing context switches and blocking the SH. On a multiprocessor, with threads actually executed in parallel, this degree of protection would be insufficient. On such a machine it would be necessary to block and unblock processors upon entry and exit of critical sections. Locking such sections would presumably be accomplished using primitives of the underlying operating system. In this case we would need to extend the LWP package procedural interface to include `AcquireLock` and `ReleaseLock` routines. Notice that minimal changes would be required in the RPC subsystem since the majority of such critical sections are encapsulated by the LWP interface.

References

- [1] B. N. Bershad and H. M. Levy, "Remote Computation in a Heterogeneous Environment", TR 87-06-04, Dept. of Computer Science, Univ. of Washington, Seattle, WA, June, 1987.
- [2] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz, "Interconnecting Heterogeneous Computer Systems", *IEEE Transactions on Software Engineering* Vol. SE-13, pp. 880-894, August, 1987.
- [3] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems* 2,1, pp. 39-59, January 1984.
- [4] A. D. Birrell, E. C. Cooper, and E. D. Lazowska, "SRC Remote Procedure Calls", Digital Equipment Corporation Systems Research Center, June 1985 (internal specification).
- [5] T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, G. L. Wyatt, "The Network Computing Architecture and System: An Environment for Developing Distributed Applications", *Proceedings of Summer Usenix*, pp. 385-398, June, 1987.
- [6] J. Q. Johnson, "XNS Courier under UNIX", Cornell University, March 1985.
- [7] J. Kepecs, "Lightweight Processes for UNIX Implementation and Applications", *Proceedings of Summer Usenix*, pp. 299-308, June, 1985.
- [8] J. Roseberg and L. Raper, *LWP User Manual*, CMU-ITC-84-037, January, 1986.
- [9] M. Satyanarayanan, *RPC2 User Manual*, CMU-ITC-84-038, January, 1986.
- [10] M.F. Schwartz, "Naming in Large Heterogeneous Systems", Ph.D. Thesis, Department of Computer Science, University of

Washington, to appear.

- [11] M. S. Squillante and D. Notkin, "A Mail System for Local, Heterogeneous Environments", TR 87-12-01, Dept. of Computer Science, Univ. of Washington, Seattle, WA, December 1987.
- [12] Sun Microsystems, Inc., *Remote Procedure Call Protocol Specification*, January 1985.
- [13] Sun Microsystems, Inc., *External Data Representation Reference Manual*, January 1985.
- [14] A. Tevanian, Jr., R.F. Rashid, D.B. Golub, D.L. Black, E. Cooper, and M.W. Young, "Mach Threads and Unix Kernel: The Battle for Control", *Proceedings of Summer Usenix*, pp. 185-197, June, 1987.
- [15] "Courier: The Remote Procedure Call Protocol", Technical Report XSIS 038112, Xerox Corporation, December 1981.
- [16] *Pilot Programmer's Manual, Version 11.0*, Xerox Corp., May, 1984.

UNIX Systems as Cypress Implets

Douglas Comer
Thomas Narten
Computer Science Department
Purdue University
West Lafayette, IN 47907

ABSTRACT

Cypress is a low cost, leased-line based packet-switched network that forms part of the DARPA Internet. The successful prototype Cypress network is a result of exploring ways to provide low-cost Internet connections. Cypress technology provides a flexible method for connecting sites to the Internet, at a fraction of the cost of conventional connections. The key ideas behind Cypress include consolidation of functionality, use of off-the-shelf hardware and software, and protocols optimized for best-effort delivery of datagrams.

This paper describes the Cypress implementation in Unix systems. It discusses the choice of Unix as well as the advantages Unix offers in building a network monitoring system. The paper focuses on how Cypress software manages a set of serial lines at the network level and its implication on the design of the subnet. In addition, we discuss the novel Cypress protocols with their conceptual separation of packet type and handling. We give the Unix implementation of the protocols, showing how a table driven kernel implementation yields fast, low overhead packet switching, and how user level processes provide a powerful paradigm for writing utilities that monitor the network.

Introduction

Cypress is a leased line based packet-switched network that forms part of the DARPA Internet. Point-to-point leased serial lines interconnect small multifunction minicomputers called *implets*. Implets connect to one another over permanent lines forming a fully connected network. An implet is placed at each subscriber's site where it attaches to a local area network (LAN) such as an Ethernet. Hosts using Cypress communicate using the DARPA protocol suite, popularly known as TCP/IP.

At subscriber sites, Implets function as Internet gateways. They route datagrams from the site's LAN onto Cypress and vice versa. One implet connects to a site that also connects to a backbone network such as NSFnet or ARPANET. Cypress forwards traffic between Cypress sites, or between sites and the backbone network.

Cypress technology offers sites flexible options for interconnecting geographically separated networks. For instance, LANs residing in different buildings at an organization or university connect over local phone lines. Such lines are inexpensive and frequently already exist. Cypress uses the same lines that connect terminals to remote machines. For geographically isolated sites, Cypress uses leased lines to connect to larger sites, which may in turn connect to a backbone network. Line cost runs proportional to the speed selected, small sites typically use low speed lines. Finally, the use of high speed lines enables

Cypress itself to be used as a backbone.

This paper describes the successful prototype Cypress network. At present, nine sites located across the country comprise the network, with several of them using Cypress as their primary Internet connection. Section 2 discusses our motivations and goals in building Cypress, and our selection of UNIX systems as implets. Section 3 describes the novel Cypress protocols and section 4 gives the Unix implementation of them. The prototype network and its performance is presented in section 5, and the final section draws conclusions from our experience.

Motivations and Design Decisions

The main goal of Cypress is to explore ways to provide Internet connections at a fraction of the cost of conventional connections. We have insisted from the beginning, however, that low cost does not mean reduced functionality. Limiting a site to only a few high level services such as mail or file transfer, prohibits that site from participating fully in the Internet community, making it a "second class citizen". Instead of providing specific services, Cypress provides the base on which all high level services can be built. Cypress uses the DARPA Internet Protocol (IP)¹ as a base, a standard that already provides interconnection between thousands of hosts at hundreds of sites. Cypress sites gain immediate use of many existing applications, including mail, file transfer, and remote login, or can create new ones

that communicate with local and remote hosts.

The need for low cost connections is especially important for small or remote sites. Scientists at workstations require access to the computing and communications resources of the Internet in order to conduct research.^{2,3} High bandwidth, cross-country networks like the ARPANET and NSFnet form the main arteries of the Internet. However, the cost of direct connections to these networks is larger than many sites can afford, and the level of service provided exceeds the basic requirements of many small sites. We view Cypress as a capillary system for reaching such sites.

Cypress technology provides a framework for creating capillary systems. Although we have constructed a fully connected prototype network, our goal is to develop a variety of techniques that make it possible to establish multiple Cypress networks, each connecting to the Internet at different points. Organizations such as NSF or CSNET will be able to provide service to their member sites by establishing Cypress networks centered at existing ARPANET or NSFnet sites.

Site Connection

One of the important ideas in Cypress is that implets communicate with hosts at subscriber sites over a local area network. In addition, Cypress implets are used only by Cypress, and are not available to execute user's programs, even though they reside at the user's site. One benefit of using a LAN to connect to Cypress is the loosely coupled nature of the connection. Second, communication between Cypress and a site's machines uses standard hardware and protocols. To use Cypress, a host forwards IP datagrams to the implet over the LAN. If the site already uses the TCP/IP protocols for communication between local machines, no changes to host software are required. Third, connecting to a site's LAN gives each host at the site fair and equal access to the implet. Fourth, Cypress availability is not tied to the normal running of any host at the site. Implets are not subject to the service outages associated with timeshared hosts. Finally, the loosely coupled nature of the LAN connection allows us to change hardware and software easily with only minimal effect on the site.

Management of Serial Lines

We decided that Cypress should be a fully connected *physical network* rather than a set of point-to-point links visible to higher layer protocols. Our approach offers several advantages over the latter approach, exemplified by the Unix serial line IP (SLIP) implementation.⁴ First, the Cypress line status monitoring mechanism is independent from that used by the higher layers, allowing us to experiment with different algorithms and new protocols. Second, Cypress has its own link-level protocol with encapsulation. Instead of using higher layer protocols like IP, the line status monitoring functions use the optimized

Cypress protocols directly, requiring less line bandwidth to perform the same function. For instance, The minimum size of an IP datagram without data is 20 octets (8 bit bytes). The header on a Cypress frame is only 2 octets. Finally, lower overhead protocols make it possible to respond more quickly to changes in topology. Status information can be sent more frequently if the bandwidth each update uses is smaller.

The set of lines and implets comprising Cypress are presented to higher layer protocols as single entity. Rather than treating each line as a distinct network (the approach taken by SLIP), all lines are viewed as part of a single network. Hiding the internal structure of Cypress from higher layers reduces the complexity of the higher level routing mechanism. In particular, all routing information about Cypress implets can be contained in a single network routing entry.

Cypress Implets

Minimizing cost strongly motivated the design of Cypress implets. Although it is tempting to think that specially designed hardware and software would meet this goal by avoiding the expense of unnecessary features, economy of scale makes general purpose machines more attractive. Cypress consolidates functionality into a single hardware package. Multifunction implets function as packet switches, Internet (IP) gateways, and run user level processes.

At the lowest level, implets function as store-and-forward packet switches, receiving frames over leased lines, queueing them temporarily, and forwarding them on towards their final destination. At the second level, each implet functions as an Internet gateway,⁵ accepting packets from the LAN and routing them onto Cypress and vice versa. An implet must correctly route packets to any address in the Internet, and it must propagate routing information to the Internet core gateways. At the third level, each implet functions like a host computer capable of executing user processes. It executes processes that monitor network traffic, log errors, and maintain routing tables.

The advantages of consolidating IP-level gateway routing and link-level packet switching becomes most evident when considering a subscriber's site. Instead of having two computers, one for packet switching and one for IP gateway routing, the subscriber site needs only one. Instead of having a special-purpose interface to connect the packet switch and gateway, the connection is made in software inside the implet. Unix provides the framework for cooperation between functions.

Selection of Unix

We chose to use Unix systems for Cypress implets. First, we wanted to begin with an existing, supported operating system instead of starting from scratch with implet software. The major advantage of modifying an existing operating system is that it provides powerful abstractions like processes and a

framework for building device and network interfaces. A second advantage is that by using an existing system that understands Internet protocols, we avoided much of the time and expense involved in implementing and debugging standard protocol software, allowing us to concentrate on Cypress issues. Unix is particularly attractive because of its extensive use within the Internet research community. Finally, a conventional operating system allows different computations to be executed at different priorities. Within the kernel, high priority, interrupt driven device drivers transfer data between memory and communications devices, restarting output on devices as soon as they become idle. Keeping devices busy is especially important when using slow communications lines. Kernel routines, executing at lower priority, process incoming Cypress frames, accepting them from device drivers and routing them to output lines or higher layer protocol software. User level processes provide a well understood paradigm for writing programs that monitor the network and maintain routing tables.

An abundant supply of vendors marketing low cost Unix systems keeps prices competitive and the relative portability of Unix to different hardware architectures provides a large potential hardware base for Cypress systems. Indeed, many sites already have Unix systems, reducing or even eliminating the cost and delay of appropriating hardware for new Cypress sites. Finally, the large vendor support for Unix systems allows us to migrate to newer hardware technologies as they become available.

In the prototype network, Cypress uses low speed 9.6 kbps serial lines. Although low speed lines provide reduced performance, substantial cost benefits result. In particular, a large variety of 9.6 kbps serial line devices and modems interoperate with one another using standard protocols. Second, Cypress uses the serial line devices supplied with most Unix systems or that can be purchased at little additional cost. Higher speed (e.g. 56 kbps) lines require special hardware devices that are considerably more expensive and less flexible. Devices for high speed lines often run special purpose link-level protocols in firmware, requiring both ends of the communications line to use identical hardware. Finally, leased line costs run close to an order of magnitude more for 56 kbps lines, whereas 9.6 kbps lines provide a sufficient level of service for many sites.

Topology

Initially, we expected Cypress networks to be vine-like with a single node attached to the Internet and other nodes attached in a long chain. The origin of the growing-vine topology comes from BITNET.⁶ The idea is simple: each site leases a line to the nearest existing site, keeping costs to a minimum.

Two things changed since the early days of Cypress, however, that make the long vine approach less appealing. First, as existing vine structured networks grew and became more heavily used, their performance grew worse. Second, with the expansion of

the ARPANET and the advent of NSFnet and NSF regional networks, it became apparent that instead of a single Cypress network, it would be more economical to have many, small Cypress nets each covering a geographical area. Each Cypress network is centered at a *hub* site, which also connects to a backbone network. Cypress sites cluster around the hub site, with only a few serial line hops separating a subscriber site from the hub site. We refer to the hub and spoke topology as a *cluster*.

Cypress Protocols

Treating all of Cypress as a single physical network has several consequences. First, not all implets connect directly to one another, requiring a mechanism for sending datagrams through implets that are along a path to the destination, but are neither the source or destination. Cypress solves this problem by using a specially designed link-level protocol with encapsulation. Cypress frames contain a small, two octet header of control information, and a data portion containing the encapsulated packet of a higher layer protocol. Upon entering the network, Cypress encapsulates frames for transmission through the network. Once inside the network, frames are forwarded towards their destination based solely on the information in the header. Only upon reaching the destination implet is the data portion extracted and passed to higher layer protocols.

Finally, an implet may connect to several others. When sending frames, Cypress needs Cypress level routing tables to select the proper output line leading to a frame's ultimate destination.

Economical Protocol

From the start, Cypress was designed to operate over low bandwidth connections, and to not depend on large amounts of processing power. The latter decision was made because implets needed to perform multiple functions, and we wanted to make sure that sufficient processing power was available for all of them. One of the consequences of minimizing processing power was that we decided not to use data compression in our prototype.

To accommodate slow speed lines, we designed the link-level protocol to have minimal overhead. First, the protocol is optimized for transport of IP packets because the network was designed for exactly that purpose. In the current protocol, only two octets of header accompany each link-level packet. No checksums or parity bits are used on link-level packets that carry IP traffic. Second, the protocol uses best-effort delivery with no attempt to detect or recover from transmission errors. The major advantage of best-effort delivery is that the link-level protocols require no checksums and introduce no additional delays. If a packet is dropped, higher-level protocols like TCP detect the problem and retransmit. Third, the protocol uses encapsulation.

One important benefit of encapsulation is the

ability to simultaneously transmit packets of multiple higher level protocols. In addition to carrying IP datagrams, special Cypress control packets are used for network maintenance and control. Another important benefit of encapsulation is that once inside Cypress, packets remain encapsulated until they reach the destination implet. When an IP packet enters Cypress, the implet computes its destination and encapsulates the entire IP packet in a Cypress frame, for transmission from one implet to another. When the packet reaches the last implet along its path, that implet extracts the IP packet and routes it outside Cypress. A major advantage of encapsulation is that it eliminates recomputing IP header checksums at each packet switch, thereby reducing delay and taking less processing power from the implet.

Frame Header

To keep Cypress flexible and amenable to change, the protocols have separate *packet type* and *packet handling* fields (see figure 1). As with most protocols, the type field specifies what the packet contains. The most commonly used packet type specifies that the packet carries an IP datagram encapsulated in it. Another packet type is used to carry Cypress control messages that implets exchange for network maintenance and control. For example, one control message type carries a message that requests an implet echo back the data portion of the packet, allowing an implet to test the functioning of other implets and the lines connecting them.

The packet handling field specifies whether the packet should be routed directly (used for conventional transmission), sent using reverse-path forwarding (used to efficiently propagate a single message to all implets),⁷ flooded, or sent across one link to an adjacent neighbor implet. Cypress uses reverse-path forwarding, flooding, and neighbor handling only for special packets.

Routing Cypress frames is extremely efficient, requiring the execution of only a few machine

instructions. For example, when routing direct packets, the eight bit destination implet id indexes into a routing table. Each table entry consists of an output line number, an error indication if the id is invalid or unreachable, or "this host" if the id corresponds to the local implet.

Frames with handling type reverse path forward contain the implet id of the frame's source. If the line on which the packet arrived is the same one that would be used to send a frame to that source, the packet is accepted and also sent out on all lines (except the one on which it arrived). If the frame arrives on a line other than the one used for sending direct frames to the source of the received frame, it did not arrive along the shortest path to the destination and is discarded as being a duplicate copy. Reverse path forwarding is an efficient way to send frames to all implets in the network. If the routing tables of each implet are optimal and consistent with each other, reverse path forwarding uses the minimal amount of line bandwidth needed to reach all sites. Cypress uses the reverse path forward handling type for sending IP level broadcast datagrams.

Neighbor handling is used to communicate with directly connected neighbor implets. The important benefit of neighbor handling is its independence from routing tables. Cypress neighbor frames are sent out on the specified line, and an implet always accepts received neighbor packets. Implets never forward neighbor frames to other implets. Neighbor handling allows an implet to send a "who are you request" to each of its neighbors before it knows their addresses. In order to use direct handling, the destination implet must be known. In addition, because neighbor handling bypasses the Cypress routing function, routing information can be collected without using the routing mechanism itself. To guarantee that an implet never misinterprets scrambled packets as control messages, the Cypress protocols specify that all control packets carry a checksum (see figure 2).

In most networks, packet handling is determined

Header				Data
type (2 bits)	handling (2 bits)	hop count (4 bits)	implet id (8 bits)	...up to 576 bytes data...

Figure 1: Cypress frame format showing the 2 octet header and data portion.

octet 0	octet 1	octet 2	octet 3	octets 4-5	data
usual	usual	msg. code	source	16-bit checksum	...

Figure 2. The format of a Cypress control message frame. Octet 0 contains the handling, type, and hop count fields, and octet 1 contains a destination or source id, as with IP packets. Octet 2 contains a code that identifies the control message. Octet 3 contains the source machine's id (independent of octet 1), and octets 4 and 5 contain a 16-bit checksum of the entire packet.

by the packet type. By contrast, the Cypress protocol specifies that an implet should act on the handling field first, and only examine the type field if the packet is destined for the implet itself. When it determines that a packet is destined for itself, the implet examines the packet type field, discarding the packet if it does not understand the specified type.

Separating the type and handling field, and following the algorithm described above gives Cypress flexibility and allows experimentation with new packet types and protocols not normally possible. To conduct an experiment, one invents a new packet type, *P*, and arranges for an arbitrary pair of implets, *A* and *B*, to use the new type. *A* and *B* can exchange packets of type *P*, even if the packets must pass through other implets that do not understand type *P*.

Unix Implementation

One consequence of using slow speed lines is that transmission delays are significant. To minimize delays, Cypress implements packet-switching in the kernel, using a table driven routing paradigm. A significant amount of packet switching consists of Cypress frames received on one line and sent on another. A kernel implementation saves the cost of context switching and copy operations over an implementation in processes. Less time critical functions, such as building routing tables and monitoring the network, are done outside the kernel in user processes.

User Level functions

User level programs executing at low priority construct the routing tables used for packet switching. *Ioctl* system calls install new tables into the kernel. Constructing tables outside of the kernel results in several advantages. First, packet switching activity takes precedence over the computation of routing tables, preventing the computation from delaying switching. Second, the building of routing tables uses spare CPU cycles, instead of competing for CPU cycles needed by the functions handling packet switching. Finally, user level processes are easier to write and debug, and run in a private, protected address space. Furthermore, they can be stopped and restarted without requiring the implet to be rebooted, an operation that would stop packet switching activity for a significant amount of time.

Another advantage of building tables in user processes is that kernel routing tables can be updated in a single atomic operation. While the kernel is switching packets using its own table, the user process constructs a new or updated table at its leisure. Once the new table is complete, a single atomic update installs it into the kernel. Installing new tables in an atomic update prevents the kernel from using inconsistent routing information in which some entries are based on old information, and others on new.

In the current prototype, a route server running on the well known implet *cypress1* builds routing tables. The server maintains network topology and

constructs routing tables using a shortest path first (SPF) algorithm. One interesting aspect of routing is the method implets use to gain initial tables. At the time implets are initially installed at a site, line 0 is configured to lie on the shortest path to *cypress1*. The route server resides at a well known TCP port, waiting for route requests. At boot time, a process running on the implet installs a route for *cypress1* through line 0. It then opens a TCP connection to the route server and retrieves a current table. Upon successful retrieval, a copy of the routing table is cached in permanent storage. That way, if the route server cannot be contacted for any reason, implets have a backup copy of recent routes.

Each implet monitors connectivity with each of its neighbors. When testing the liveness of an implet, the testing procedure should not depend on Cypress routing tables. Otherwise, once a line stops functioning and the routing function stops using it, it would be impossible to determine if the line starts working again. Cypress control messages with neighbor handling solve this problem. Neighbor handling sends a Cypress frame out on the specified line. It is not even necessary to know the identity of the implet at the other end of the line.

The Cypress line monitoring process sends Cypress control echo messages on each line every 15 seconds. A *k* out of *n* reachability algorithm⁸ determines the reachability of neighboring implets. If an implet receives *k* responses to the *n* most recent requests, the line is declared up, and Cypress routes frames over it. Whenever *k* of *n* responses are missed, the line is marked down, and the routing function no longer uses it. Use of this algorithm prevents rapid fluctuations in link status, yet insures that a line that stops functioning stops being used. For instance, it is not uncommon to have a burst of line errors lasting a few seconds or less causing a momentary loss of carrier. Discarding packets during the short burst is preferable to marking the line down, because higher layers may be notified when a link no longer functions and the implets or subscriber sites behind it become unreachable. Cypress currently uses 3 and 4 for values of *k* and *n* respectively.

Monitoring line status using Cypress control echo packets provides a reliable mechanism for testing communications lines and the implets at both ends. When a link no longer functions properly, higher level protocols can be notified when they attempt to send datagrams which travel across those links. For instance, IP returns ICMP destination unreachable⁹ messages to the originator of a datagram. Besides not sending frames across lines that are marked down, Cypress discards frames arriving on those lines unless they contain Cypress control messages. Discarding incoming frames insures that the line will not be used at all if the link fails in one direction only. Our experience shows that when a phone line fails, it often still transmits correctly in one direction. One-way only links cause considerable confusion to higher level

routing protocols such as routed,¹⁰ that accept datagrams without verifying that the source of the datagram can be reached.

To monitor packet switching activity on remote implets, Cypress uses a server based monitoring system. A client program running on a local machine (not necessarily an implet) opens a TCP connection to

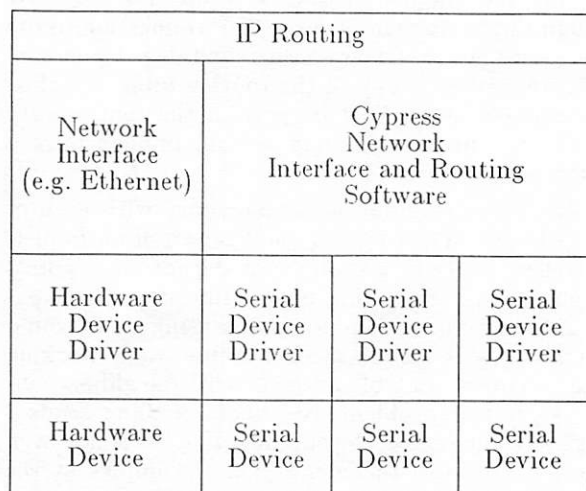


Figure 3. Cypress packet switching code resides between the IP network layer and serial line device drivers. Although Cypress is a single network, a given implet may have several physical devices managed by the Cypress network interface code.

a server running on the implet being monitored. The server sends the client statistics collected by the packet switching functions, which client software displays in a graphics oriented manner. To keep the display up to date, the server sends updates every five seconds. Statistics collected and displayed include per-line packet counts, characters transferred, cpu utilization, and queue lengths.

The server based monitoring system is extremely powerful and has worked extremely well. The choice of TCP as a standard transport protocol offers several benefits. First, when debugging network problems, the information being monitored must be transmitted correctly and reliably. Building a reliable transport protocol on top of the Cypress protocols would require a substantial amount of effort, and would not provide more functionality than that provided by TCP. Second, an implet can be monitored from any host in the Internet. The monitoring host is not required to connect to the Cypress network itself. This is important for Cypress because hosts at subscriber sites do not directly connect to Cypress, only the implet does. Finally, using a standard protocol allowed us to concentrate on the quantities being monitored, rather than on the details of how to get information from the implet to the monitor. In fact, the ease with which the monitor can be modified has encouraged us to change the server several times to collect better statistics.

Kernel

The Cypress packet switching code resides in the Unix kernel between the IP layer and hardware devices (see figure 3). The method of passing datagrams between IP and Cypress is identical to that used between IP and other network interfaces. IP datagrams routed to Cypress are passed to Cypress via an input procedure made visible to IP. Cypress places datagrams destined to IP in a shared queue. No changes to the IP code are required to use Cypress.

Cypress uses standard serial line devices for its lines. In Berkeley Unix, device drivers for terminal devices support multiple *line disciplines*¹¹ - an abstraction that allows a single device driver to perform specialized operations on a per-line basis. Most serial line devices support multiple lines per device, the line discipline mechanism allows each line to be configured independently of the others. Cypress uses the line discipline mechanism to bind specific serial lines to Cypress. A Cypress utility program opens the terminal device, and changes the line discipline to that used by Cypress. The act of changing the line discipline invokes a Cypress initialization routine that binds the hardware device to a Cypress line, and Cypress begins using it. At this point a logical line is bound to a real hardware device.

Once bound to Cypress, the line discipline mechanism handles the sending and receiving Cypress of frames from the hardware device. Complete Cypress frames ready for transmission are copied into a contiguous buffer. The device specific output routine transfers the entire buffer in a single operation. Devices supporting direct memory access (DMA) transfer interrupt the CPU only after the entire frame has been transmitted. To reduce the per-character interrupt load on input packets, device drivers use clock interrupts to poll the hardware device periodically. When the communications line is running at maximum speed, each poll retrieves many characters. When a complete frame has been received, the frame is inserted in a queue shared with the Cypress layer. Communicating with Cypress through a common queue allows the hardware device drivers to run at high priority, while the Cypress routines run only when the devices do not require CPU resources. Hardware devices run at the highest priority; when using low speed lines, we cannot afford to let lines remain idle long when frames are ready to be transmitted.

Finally, hardware device drivers use upcalls to notify Cypress of changes in device status, such as loss of carrier. The upcall is accomplished through the line discipline mechanism. Although carrier loss itself does not cause Cypress to stop using the line, logging status changes to permanent storage is important for trouble shooting and monitoring phone line quality.

User level programs access Cypress in two ways. *Ioctl* system calls into the kernel allow programs to install new routing tables, extract logged error messages, and determine the current configuration of

lines. The *raw* socket mechanism¹² provides user programs direct access to the Cypress packet switching layer. Complete Cypress frames are handed to the Cypress layer, which routes and queues them on the appropriate output line. All Cypress control packets addressed to the local implet are passed to user processes through the *raw* socket mechanism.

Prototype Network

To test our ideas, we built and operated a prototype network centered at the Computer Science Department of Purdue University in West Lafayette, Indiana. The prototype has been operational since November, 1985 with traffic between the prototype and the rest of the Internet passing to the ARPANET through an Internet core gateway, also at Purdue. From West Lafayette, lines run east to Massachusetts, west to California, southwest to Arizona, and north to Chicago, connecting the following sites:

CSNET Coordination and Information Center	BB&N Inc., Cambridge, Massachusetts
Digital Equipment Western Research Laboratory	Palo Alto, California
Williams College	Williamstown, Massachusetts
Boston University	Boston, Massachusetts
University of Chicago	Chicago, Illinois
University of Arizona	Tucson, Arizona
Atmospheric Sciences Department at Purdue University	West Lafayette, Indiana
Silicon Graphics	Mountain View, California

Several of these sites rely on Cypress as their only connection to the Internet. For example, the University of Arizona's Computer Science Department has used Cypress exclusively for over two years. They routinely send mail and login to remote machines.

Hardware

Early in the design we decided to use off-the-shelf hardware and software systems. The original implets used for our prototype network were VAX 11/725 computers donated by Digital (The 11/725 uses a VAX 11/730 processor which performs at approximately 0.3 MIPS). These are being replaced by newer machines that are much faster. In particular, we have implet software running on the Digital Equipment Corporation Microvax II hardware and Sun 3 systems, and it was recently ported to the Silicon Graphics IRIS 3030. Only two 725s remain in production.

We have experimented with a variety of serial line interface hardware. Digital VAX machines use standard DMA hardware that transmits an entire

packet with only one interrupt. Hardware used to drive ASCII terminals can be used at 9.6 kbps, but a special interface board is needed for 56 kbps lines. On Sun systems, Cypress uses the standard serial ports that require an interrupt per character, but the CPU is fast enough to handle the additional overhead. In fact, the Sun implets operate at speeds from 9.6 kbps through 56 kbps using the same hardware.

We started with Berkeley's 4.2 BSD version of Unix as our base system. Our implets now run 4.3 BSD, Ultrix (Digital Equipment Corporation's version of Unix), Sun OS (Sun Microsystems' version of Unix) and IRIS 3000 OS (Silicon Graphics' version of Unix). In principle, Cypress can be ported to any flavor of Unix supporting the Internet protocols, and the Unix socket interface.

Performance

Performance of the Cypress prototype has been extremely good. Because the protocols are optimized for transport of IP datagrams, the link-level introduces little overhead. During file transfers, for example, user data accounts for approximately 85% of the raw hardware line speed. Typical throughput for a file transfer across a 9.6 kbps line averages over 800 bytes per second user data. TCP and IP overhead accounts for most of the rest, with less than 1% attributable to Cypress. More details can be found in.^{13,14,15}

Conclusions and Future Directions

When the Cypress project started several years ago, it was not obvious that a best-effort style of link-level delivery over slow speed leased lines would provide reliable service, or that a single CPU would be capable of handling the processing required for a multi-function packet switch like a Cypress implet. The success of the experimental Cypress prototype demonstrates the viability of these ideas, and leads to the inescapable conclusion that it is possible to provide reliable, low-cost Internet access. In fact, advances in processor technology make it clear to us that conventional operating systems should be used in all IP gateway machines, and that monitor and control software should be written to use transport protocols like TCP to communicate.

We also learned that a hub-site topology is important. Cypress started as a way to provide CSNET with a low-cost alternative to ARPANET or X25NET,¹⁶ so we looked for a topology that incurred least cost. Part of the motivation for moving to a hub-site approach came from potential subscriber sites, who were willing to pay a little more for a line connected "closer" to the Internet. The expansion of the Internet by NSF introduced many more potential connection points, and forced us to consider alternatives to the original vine topology. With main arteries and regional networks in place, it is clear that Cypress fits in as a technology that provides capillary connections. We find it ironic that, while Cypress was started to help find ways to provide IP connections,

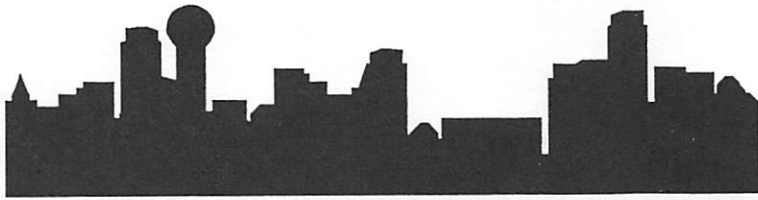
the original topology resulted from underestimating how widespread Internet connections would become.

Besides experimenting with new machines and networks, we plan to help CSNET establish a higher speed Cypress technology. Plans for upgrading Cypress to 56 kbps are already underway. 56 kbps lines have been successfully tested in a laboratory setting, and a high speed line from Purdue to Palo Alto, California is scheduled for installation in February, 1988.

In addition, we will continue to tackle new research problems using Cypress as a testbed for our ideas. For instance, congestion results when too many hosts use Cypress simultaneously. If the amount of traffic destined for Cypress exceeds the capacity of the network, datagrams must be discarded. We are experimenting with different queuing strategies, that react to congestion before datagrams must be discarded. The problem is not unique to Cypress; results will apply to any long-haul network in the Internet.

References

1. J. Postel, "Internet Protocol," RFC 791, DDN Network Information Center, Menlo Park, CA 94025 (September 1981).
2. D. Jennings, L. Landweber, D. Farber, and W. Adrion, "Computer Networking for Scientists," *Science*, pp. 943-950 (February, 1986).
3. B. Leiner, "Network Requirements for Scientific Research Internet Task Force on Scientific Computing," RFC 1017, DDN Network Information Center, Menlo Park, CA 94025 (August, 1987).
4. *Manual Page for slattach*, 4.3 BSD UNIX June, 1986.
5. R. Hinden and A. Sheltzer, "The DARPA Internet Gateway," RFC 823, DDN Network Information Center, Menlo Park, CA 94025 (September 1982).
6. Ira. H. Fuchs, "BITNET: Because it's time," *Perspectives in Computers* 3(1)(March, 1983).
7. Y. Dalal and R. Metcalf, "Reverse Path Forwarding of Broadcast Packets," *Communications of the ACM* 21(12) pp. 1040-1048 (December, 1978).
8. D. Mills, "Exterior Gateway Protocol Formal," RFC 904, DDN Network Information Center, Menlo Park, CA 94025 (April, 1984).
9. J. Postel, "Internet Control Message Protocol," RFC 792, DDN Network Information Center, Menlo Park, CA 94025 (September, 1981).
10. *Manual Page for routed*, 4.3 BSD UNIX June, 1986.
11. *Manual Page for tty*, 4.3 BSD UNIX June, 1986.
12. S. J. Leffler, W. N. Joy, R. S. Fabry, and M. J. Karels, *Networking Implementation Notes 4.3 BSD Edition*, 4.3 BSD UNIX June, 1986.
13. D. Comer and T. Narten, "The Cypress Multifunction Packet Switch," Technical Report CSD-TR-575, Computer Science Department, Purdue University (March, 1986).
14. D. Comer and G. Smith, "Sun Workstations as Cypress Implets," Technical Report CSD-TR-581, Computer Science Department, Purdue University (March, 1986).
15. D. Comer and G. Smith, "Early Cypress Performance Experiments," Technical Report CSD-TR-662, Computer Science Department, Purdue University (March, 1986).
16. D. Comer and J. T. Korb, "CSNET Protocol Software: the IP-to-X.25 Interface," *SIGCOMM 83*, pp. 154-159 (March, 1983).



Michael J. Yamasaki
NASA Ames Research Center
M/S 258-5
Moffett Field, Ca. 94035

Special Purpose User-Space Network Protocols:

ABSTRACT

Without the constraints of general purpose, standard, kernel implementations of network protocols, special purpose user-space protocols can be valuable in solving a variety of problems. The ability to carry out such diverse activities as remote login and file transfer, while remaining interoperable with a variety of machines and operating systems, connected by anything from a serial line to a satellite, under a single protocol suite may not matter much to you. It may be more important to you, for instance, to develop in relatively short order a means to move data quickly between your super-computer and mass storage on your mainframe.

Following a brief description of the computing environment in which these protocols were developed, the Numerical Aerodynamic Simulation (NAS) facility at NASA Ames Research Center, three examples of user-space protocols developed for use with HYPERchannel to communicate between a Cray 2 running UniCOS, an Amdahl 5880 running UTS/580, and SGI IRIS workstations running UNIX are described with comments about their purpose, scope, and implementation:

- i. Temporary User Backup Environment (TUBE) - a short life cycle utility, providing a file archive mechanism for archiving files from the Cray 2 to mass storage on the Amdahl 5880. An application integrating simple data base management on the Cray 2, file transfer via HYPERchannel, and tape storage and retrieval on the Amdahl 5880.
- ii. Bulk Copy (bcp) - Fast file transfer to/from Cray 2 UniCOS from/to Amdahl 5880 UTS. This protocol features large logical packet size utilizing the HYPERchannel in an efficient manner.
- iii. Distributed Library (DL) - Using the low level packet passing protocol from Bulk Copy, DL provides a stateful remote procedure mechanism with a extensible library of routines.

A particular application under development may require a communications feature which is not available using the commercially available standard products offered for this application's operational environment. In this type of situation the developer has few options. This paper discusses the "roll your own" approach: its utility and its drawbacks.

Introduction

One uses standard network protocols for a variety of reasons. In a heterogeneous network environment standard protocols offer a distribution of the "implementation workload", usually among the participant vendors. Standard protocols, offering network canonical forms for such things as byte order or data representation, simplify the task of coordinating the communication of machines of such diverse architectures as supercomputers and PCs. The solution for everything from remote login over a LAN to videotex over satellite links may be found within the confines of a single protocol suite implementation. The solution for *your* particular problem may not.

Finding yourself in a situation where a standard protocol implementation does not meet your needs, may lead you to a decision to develop a special protocol on your own. In developing special purpose protocols, conventional wisdom may not always point the path to success. Use a stop and wait protocol over sliding windows? A user-space implementation over a kernel implementation? The following is a description of several protocols developed at the Numerical Aerodynamic Simulation Systems Division at NASA Ames Research Center.

Site Description

Numerical Aerodynamic Simulation (NAS) combines the power of supercomputers, high speed networks, and high resolution graphics to provide the capability to utilize computational fluid dynamics (CFD) in modern aircraft design. CFD combines the disciplines of fluid physics, computer science and applied mathematics to simulate and study viscous flow phenomena. Distributed processing and interactive visualization play key parts in developing this capability.

In 1983, the NAS Program was initiated at NASA Ames Research Center. The objectives of the NAS Program are to:

- Provide a national computational capability, available to NASA, DOD, industry, other Government agencies, and universities, as a necessary element in

insuring continuing leadership in computational fluid dynamics and related disciplines.

- Act as a pathfinder in advanced, large-scale computer system capability through systematic incorporation of state-of-the-art improvements in computer hardware and software technologies.
- Provide a strong research tool for the NASA Office of Aeronautics and Space Technology (OAST).

With this charter, the NAS Systems Division has developed a computing environment with a high speed data network as its communication backbone, supercomputers for its computational power, graphics workstations for interactive visualization, and UNIX as its common user interface. The basic elements are depicted in Figure 1.

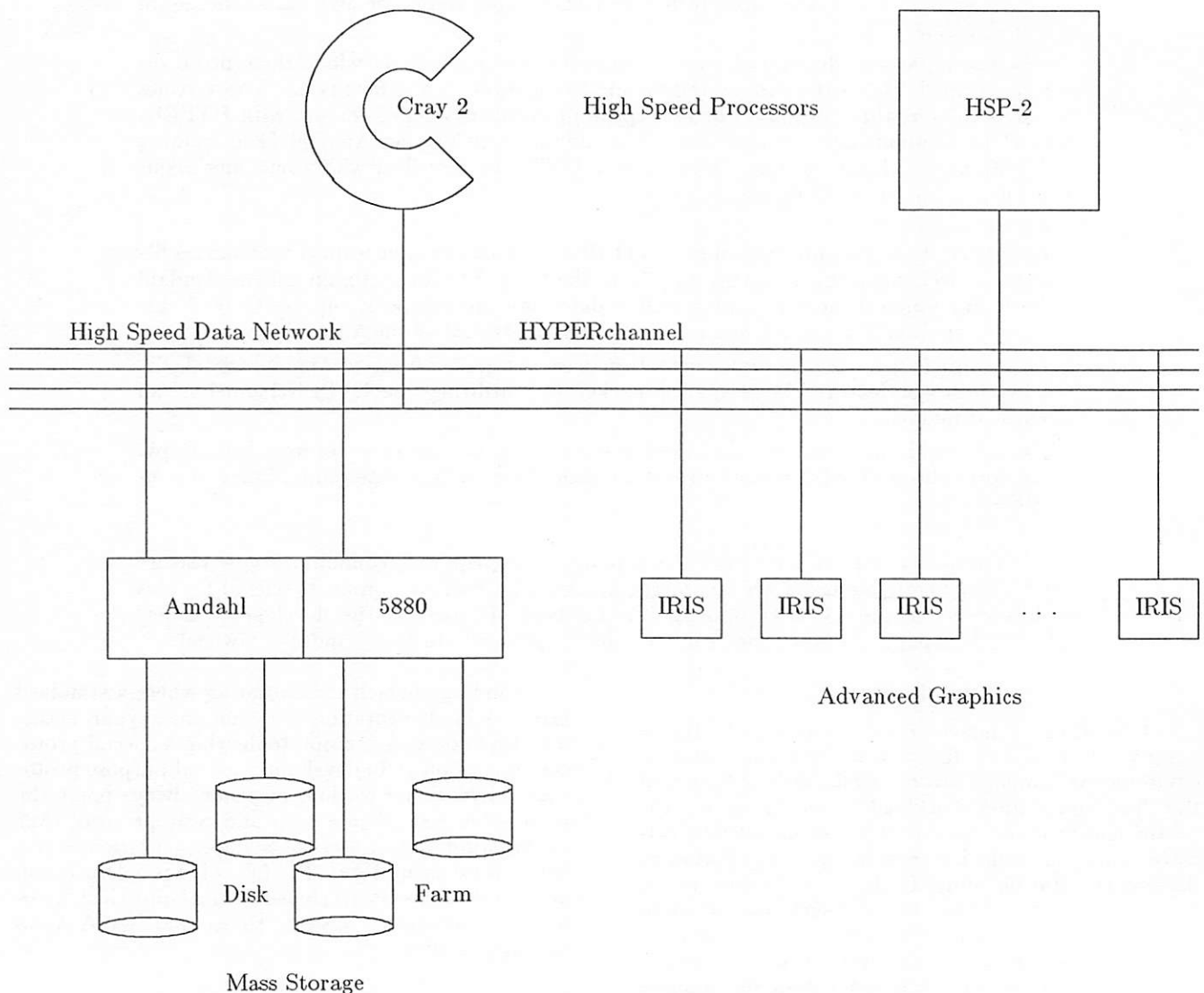


Figure 1. Basic Elements of the NAS Processing System Network (NPSN)

The High Speed Processor-1 (HSP-1) in the NPSN is a Cray 2. Currently, a second high speed processor, HSP-2, is being procured. The operating system of the Cray 2 is UniCOS, a UNIX SV derivative with various network additions from 4.2 BSD.

An Amdahl 5880 running UTS, also a UNIX SV derivative, with BSD additions from The Wollongong Group, administers the disk farm and tape library for the NPSN. This mass storage subsystem provides on line mass storage in the hundreds of gigabytes.

Silicon Graphics IRIS workstations provide advanced 3-D graphics capability, rendering 5550 Z-buffered, Gouraud-shaded 100-pixel polygons per second.

HYPERchannel Network

The NPSN high speed data network is a 4 trunk HYPERchannel network with a serial data transmission rate of 50 megabits per second. HYPERchannel Adapters provide a buffered interface between the trunks and host CPUs. The adapters implement a CSMA/CD type network with a trunk and adapter reservation mechanism and features a multiple frame logical packet. The HYPERchannel network message has two basic parts: the message proper and associated data.

The message proper can be considered a 10 to 64 byte packet header with address and control information. The first 10 bytes are used for hardware control and addressing. The remaining bytes of the message proper are for use by higher level protocols. Associated Data is optional user data which is transmitted between adapters in 4K byte quantities to an unlimited quantity. In practice the limit of the logical packet size (message proper plus some number of 4k byte data frames) is determined by the device driver.

What is important to understand about the HYPERchannel in the context of this paper is that the logical packet size can vary depending on the device driver from a nominal minimum of 64 bytes plus 4K bytes associated data to 64 bytes plus 64K bytes (or more) associated data. This range of flexibility is an important consideration when comparing user-space and kernel implementations.

Early Experiences with Special Purpose User-Space Protocols

The Cray 2 was delivered at NASA Ames in late 1985. Early versions of the operating system did not include an implementation of TCP/IP. The Cray 2 does not have serial ports in the way most machines have. Terminal access is usually through a front end machine which communicates with the Cray 2 via HYPERchannel. Current versions of UniCOS support terminal access via telnet or rlogin.

Two utilities were provided for remote login and file transfer, *cxint* and *hyft*, respectively. These were, basically, simple stop and wait protocols which operated over HYPERchannel to a VAX running

UNIX SV. They were implemented in user-space and used raw HYPERchannel devices. A raw HYPERchannel device refers to a driver interface which uses the basic *open()*, *read()*, *write()*, etc. device interface as opposed to a socket/network interface. For a small number of front end machines these protocols were quite adequate.

Remote login and file transfer are best served by general purpose standard protocols. As different types of machines proliferate on the network and internet issues come in to play, support for special purpose protocols across the whole range of the network becomes problematic. If the protocol design is simple, often the speed of implementation of a special purpose protocol is reason enough to pursue a short life cycle utility, while the longer term utility implementation proceeds more deliberately.

Temporary User Backup Environment (TUBE)

Still without an operational TCP/IP implementation for the Cray 2, disk space on the system was being rapidly utilized and it became apparent that a means to move data from Cray disk to less costly media (i.e. tape) was needed. Already a project was underway to provide mass storage management via an Amdahl mainframe, but its completion was not to be seen for over a year. Directly attachable tape drives for the Cray 2 are just now becoming available, two years hence.

So, a situation arose in which vendors' offerings did not contain a solution to the problem of tape storage for our operating configuration. Local efforts to provide an answer would not reach timely fruition and the severity of the problem could only increase. It is in this type of situation that one considers the special purpose solution. It is in this situation that the TUBE was developed.

The main features of the TUBE are as follows:

- i. File renaming - To maintain distinct copies of multiply archived files, upon storage in the TUBE a file was given a unique name and cataloged by date stored.
- ii. File splitting - Maximum file size on the Cray 2 greatly exceeds that of a 9 track tape (about 100M bytes). Current backup software does not allow a file to span more than one tape, so the files are split in transit to files no larger than 40M bytes.
- iii. Simple file transfer protocol - A stop and wait protocol using raw HYPERchannel devices. Files are transferred to one of two UTS partitions, while the other partition is being backed up to tape. The latter partition is then remade with disk gap sizes optimized for writing. This allows for an unfragmented disk with the best performance on storing a file from the Cray 2, a time critical process.
- iv. Minimal human intervention - Tape mounting is the only human part played in the process of file

renaming, file transfer and splitting, backup triggering, restore, and file transfer and reassembly.

The TUBE worked reasonably well over its production life-span of one year, storing approximately 60G bytes during that time. The basic flow sequence of the simple stop and wait protocol, upon which the higher level file transfer protocol for the TUBE is built, is outlined in Figure 2.

This protocol has the virtue of being completely controlled by the transmitting side of the exchange with the receiving side unable to initiate communication. An unacceptable data packet (e.g. bad checksum) is indicated by the receiver by simply not responding. The transmitter will retransmit on a time out and abort on excessive retransmits. The flaw in this protocol is in the case of a dropped acknowledgment. The transmitter not receiving an acknowledgment of a data packet will retransmit. The receiver having (seemingly) already acknowledged that data packet (as indicated by the sequence number) will drop that packet as a duplicate. This will continue until the transmitter aborts due to excessive retransmissions. The fix for this flaw is simple - add a retransmission of the acknowledgment for duplicates of the last acceptable data packet.

The flaw and solution are mentioned here to contrast the differences in development for a user-space protocol verses a kernel implementation. In a user environment like the NAS most machines and certainly the Cray 2 are in production mode a large majority of the time. The difference between implementing this simple change in a protocol in the normal production environment as opposed to installation and test of a new kernel for each machine in terms of real time is immense. If implementation speed is a major goal, user-space implementations of

network protocols becomes reasonable.

Bulk Copy

Conventional wisdom and scholarly studies [Zwaenepoel 1985] tell us that sliding window protocols perform substantially better than stop and wait protocols. In real life situations, circumstances may conspire to undermine this truism.

Flow control in network protocols is usually dependent on buffer space available to the communicating processes. The basic scenario in which sliding window protocols have significant performance advantage over stop and wait protocols contains these features:

- i. Buffer space available to transmit and receive data is a multiple of the maximum transmission size of the network to allow several data packets to be transmitted per acknowledgment packet.
- ii. Performance is considered for a single data stream.

HYPERchannel has the interesting property of having no specific maximum transmission size. An effective maximum is dependent on the implementation of the HYPERchannel device driver. Currently, on NAS machines this number ranges from 64K bytes (Cray 2) to 4K bytes (IRISs, VAXs). Consequently, the maximum transmission size for the HYPERchannel network can be adjusted to be equal to the buffer space available. Doing this would cause a situation in which the entire window could be transmitted in a single data packet. In other words, given a network which transmission granularity is large while maintaining a reasonably low error rate, sliding window protocols become stop and wait. So, in order for sliding window protocols to provide a significant performance advantage over stop and wait, the window

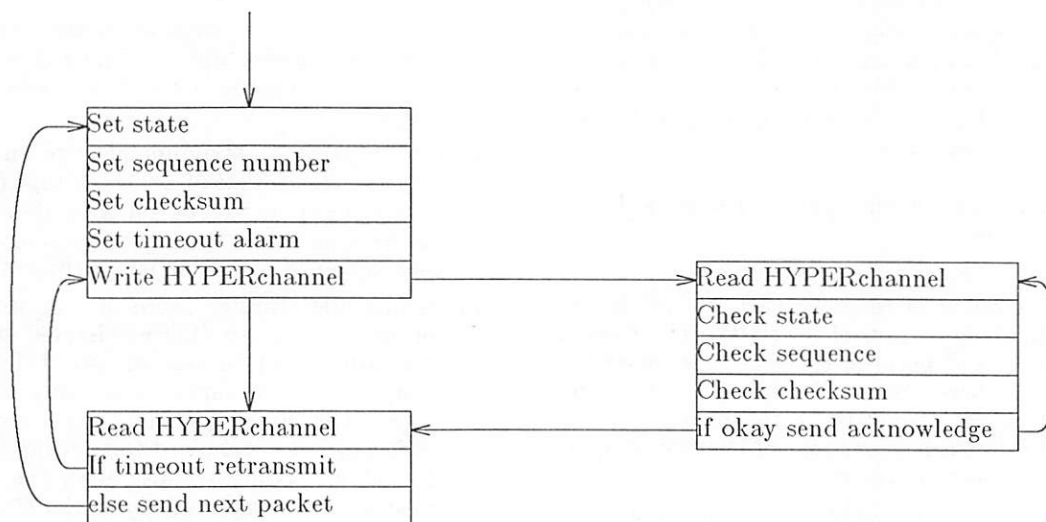
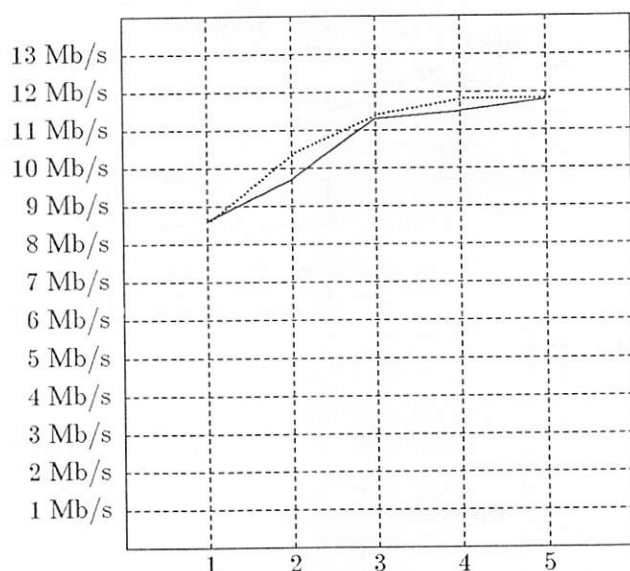


Figure 2. Simple Stop and Wait Protocol

must be a multiple of the maximum transmission size of the network to allow several data packets to be transmitted per acknowledgement packet.

The second condition which must exist for sliding window protocols to offer a significant performance advantage over stop and wait protocols is that performance is considered for a single data stream rather than multiple data streams. When the aggregate transfer rate of multiple streams over a stop and wait protocol is compared with the single stream transfer rate over a sliding window protocol, it appears that a sliding window protocol allows a single stream to acquire a larger percentage of the potential aggregate rate. But the aggregate rate is not appreciably increased.

Figure 3 compares the performance of a single data stream over a sliding window protocol with successively larger windows with multiple streams of a stop and wait protocol operating between a Cray 2 running UniCOS and an Amdahl 5880 running UTS. Both protocols are user-space protocols over raw



For solid line: Number of simultaneous data streams (stop and wait protocol)

For dotted line: Window size in number of 56K data packets - single data stream (sliding window protocol)

Figure 3. Performance Comparison

Aggregate stop and wait vs. Single sliding window HYPERchannel devices.

For the maximum transmission size used (56K bytes), 12.5M bits/second is the maximum rate achieved by sending packets without acknowledgment between the Cray 2 and Amdahl 5880. This can be considered the effective bandwidth over HYPERchannel (single adapter pair) between these two machines. The dotted line in Figure 3 shows that by increasing the size of the window in a sliding window protocol, a

single data stream can utilize almost the entire bandwidth of the network. The same effect can be seen in the aggregate rate with multiple data streams for a stop and wait protocol.

This result indicates that in environments where multiple concurrent transfers are the normal case, sliding window protocols may not offer a significant performance advantage. Given the NPSN environment and other considerations, it was decided that the implementation overhead of a sliding window protocol was excessive for the return value and a stop and wait protocol was implemented for bulk data file transfer.

Distributed Library

With the Cray 2 used for its computational power and the IRIS workstation used for interactive visualization, distributed processing plays a key part in NAS research. The development of a bulk data transfer protocol for efficient use of HYPERchannel bandwidth led to the desire to utilize this protocol for a remote procedure call (RPC) mechanism.

In the development of a RPC utility it would be desirable to minimize the programming effort involved in distributing a program. A RPC mechanism, where the procedure executes on the remote machine and returns some value to the local machine, is used to diminish the extent to which the user is involved with the specifics of the distributed system. Side effects or the manipulation of data structures global to the procedure becomes an interesting question for remote procedure calls.

The development Distributed Library (DL) is an attempt to allow for the saving of state between remote procedure calls. This idea is implemented by the insertion of a "layer" between the client and the remote procedure. The differences between a stateless RPC and the stateful DL is illustrated in Figure 4.

In the DL Model, the client process initiates an environment on the remote machine by posting a request to a dispatcher. The dispatcher coordinates the rendezvous between the client process and a DL server process. The DL server process (DL Back End) is the repository of state information from which DL routines can be called.

Two special DL procedures are required, remote memory allocation and data transfer. For remote memory allocation, a command is sent from the client to the DL server to allocate memory space (i.e. the command *dlmalloc()* by the client user process causes a *malloc()* to be called by the server process). A *memory descriptor* is returned to the client for identification purposes. The data transfer procedures are used to synchronize data in a client buffer with a server buffer (obtained by *dlmalloc()*) with either buffer updated.

There are two major advantages to programming using the DL approach as opposed to a RPC

approach:

- i. RPC tends toward complexity, while DL tends toward simplicity.
- ii. The major programming effort in writing an application using DL is centralized while with RPC it is distributed.

The first advantage is best described by example. On the remote machine one might want to do these things:

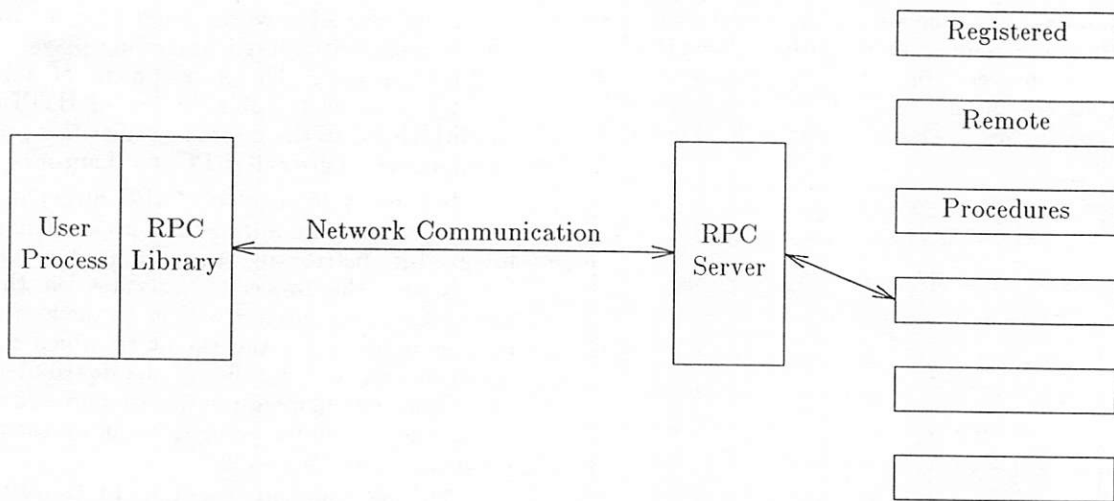
Open a file.
Read from that file.

Manipulate the data.
Open a new file.
Write to that new file.

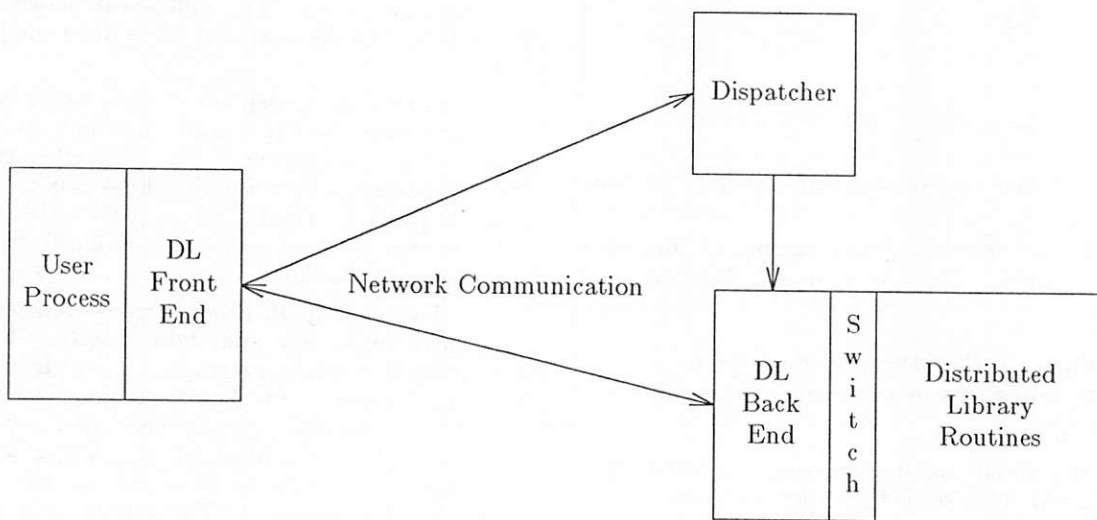
With the RPC paradigm the above would be accomplished in one RPC. With DL however, each listed item would be accomplished by a separate call. The DL remote procedures in this case would be simple system-utility-like calls. Context such as file descriptors is kept by the DL server.

Flexibility is added to remote actions in the RPC paradigm by adding complexity to the remote procedure. Flexibility is indigenous to the DL paradigm, promoted by the ability to save state.

The second advantage is related to the first



Model of State-less Remote Procedure Call



Model of State-ful Distributed Library

Figure 4. Remote Procedure Models: State-less and State-ful

while being controlled from the user's workstation.

The main feature of the user interface is that it allows the user to dynamically create and modify the network configuration. Nodes and links can be created and destroyed, and the processes being run on the individual nodes and the functions controlling the links can be changed with the user interface. This is done with mouse button clicks in the network window, where a diagram of the current network is displayed. Clicking the left mouse button creates a new node at the current mouse position. Links are created by positioning the cursor over an existing node, then dragging the mouse with the middle button down until it is over another node.

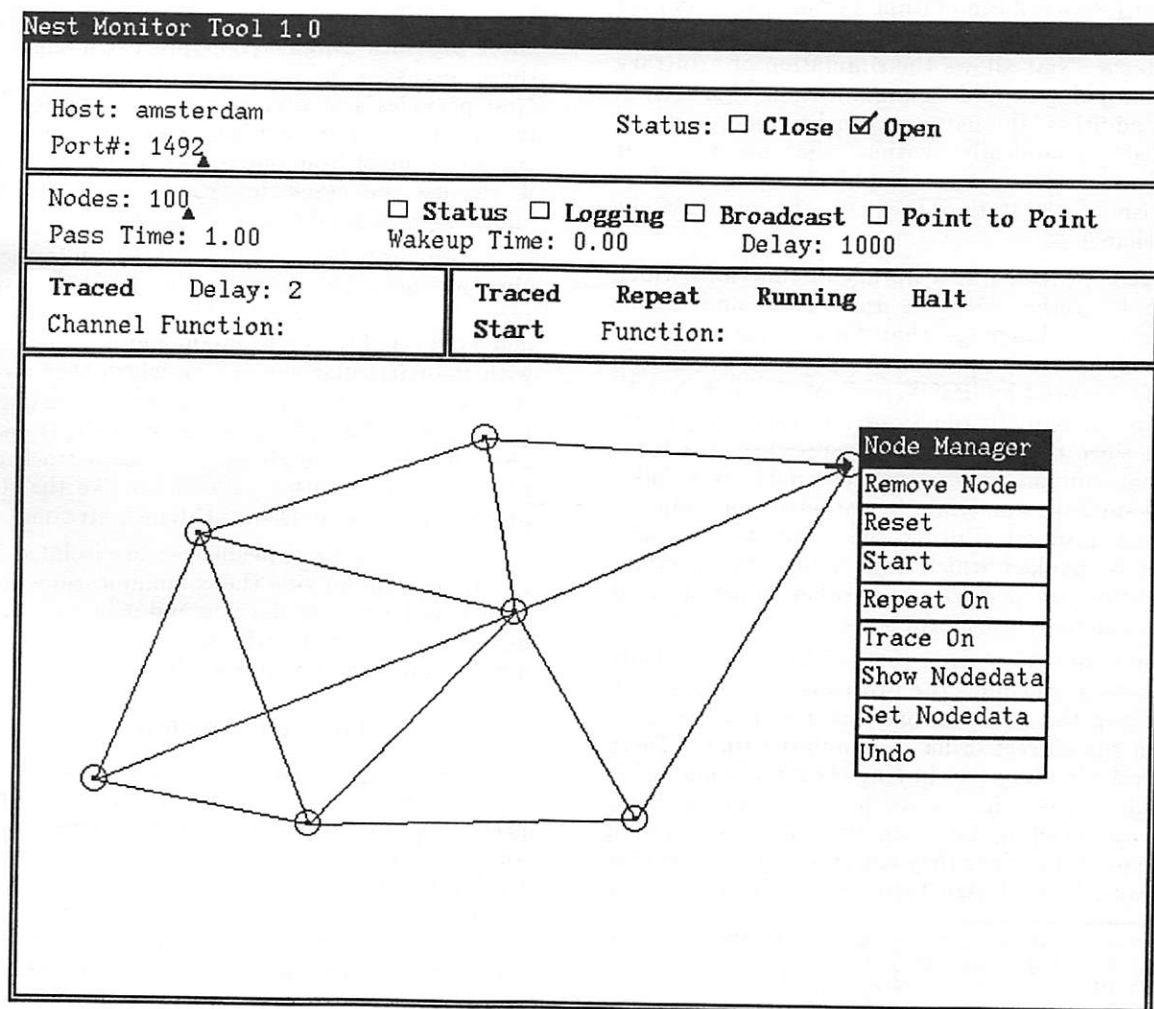
Deletions and other modifications can be made using the pop-up menus which appear when the right mouse button is pressed. When this button is pressed while the cursor is on or near a node, a menu appears with items which allow the user to delete or modify the node, as in the illustration. It is also possible to display more detailed information in a panel above the network display, where it can be modified. A similar pop-up menu appears when the right button is pressed while the cursor is over a link. If the right

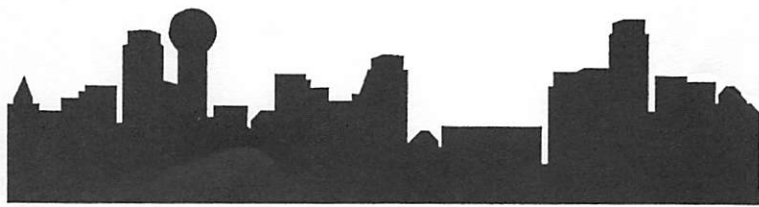
button is pressed while the cursor isn't over any node or link, another menu is called up, with an item to undo recent deletions, as well as one which sends a message to the simulation containing all the changes to the network which the user has made.

A panel above the network display allows the user to control simulation parameters, including the granularity of the simulation phases, whether the network supports broadcast messages, and the default delay for messages being sent through links. There is also a panel which allows the user to specify the simulation which the user interface is connected to, whether other user interfaces can control the simulation, and permits the entire simulation to be paused (perhaps to be resumed at a later time).

An Example Nest Application

Having looked at the basic user interface to Nest, we will turn to the programmer interface (*i.e.* what someone writing a simulation, or prototyping a distributed system, needs to know in order to use Nest). In the classic Unix tradition, we will use a variant of the famous "hello, world" program to demonstrate the





Nest: A Network Simulation and Prototyping Tool

David F. Bacon
IBM T.J. Watson Research Center

Jed Schwartz
Yechiam Yemini
Columbia University Department of Computer Science
520 West 120 Street
New York, NY 10027
{dupuy, jed, yemini}@cs.columbia.edu

ABSTRACT

This paper describes Nest, a testbed which provides a simulated network environment for developing and analyzing distributed systems and algorithms. Nest has a number of interesting features, including a transparent implementation of lightweight processes under UNIX, and a distributed monitoring facility with a graphical user interface.¹

Introduction

Nest (Network Simulation Testbed) is a tool for simulating and prototyping distributed algorithms and systems. Nest allows the simulation of arbitrary network topologies and communication characteristics. In addition, the network can be configured and controlled dynamically within the program, or through a mouse-driven graphical user interface which displays the state of the network and allows the user to change it.

Nest is provided as a library of functions which are linked together with the user's code, and can be used with any language that follows the standard stack discipline; this approach to network simulation has also been used by [Bac87], [Cook87], and [Xu87]. There are no built in functions for statistics gathering, but since the user can easily program the functions that run on nodes and pass data over links, relevant statistics can easily be gathered. In addition, at regular intervals during the simulation, a user function is invoked which can change the network configuration or perform any other function that requires a globally consistent state.

The entire simulation runs within a single Unix process. Nest schedules the processes associated with the nodes in the network, and ensures that messages arrive in the correct order in simulated time. There are several advantages to having the simulation inside of a single process: first, users need not concern themselves with encoding data structures for transmission over the network, since they can simply pass a pointer to a heap-allocated structure. Secondly, debugging

tools like dbx which only operate on a single Unix process can be used. Finally, it is possible to simulate systems with large numbers of processes, since Nest in effect provides a lightweight process mechanism like those discussed in [Kepe86] and [Libe87]. However, Nest provides a greater degree of transparency, since it does not require explicit co-routine functions to transfer control from one process to another, nor does it require the stack for each process to be pre-allocated or of fixed size.

Nest provides a communications interface to an abstract network through the functions *sendm()*, *recvrm()*, and *broadcast()*. By default, messages are delivered reliably in a fixed amount of time associated with the particular channel on which they were sent. However, users may provide their own channel functions to model arbitrary behavior of the transmission medium. In fact, each channel has a stack of functions associated with it, somewhat like the stackable protocol modules in Dennis Ritchie's streams.

Since network dependencies are isolated in a few functions that provide the communications interface, Nest may be used to develop and debug prototypes of network applications which can then be run on real networks with only minor modification.

The Nest User Interface

Nest applications are controlled through a separate user interface program. This user interface presents a graphical display of the network, as well as a panel of controls for simulation operations (see Illustration 1). It communicates with the simulation using a TCP/IP connection, and multiple user interface programs can connect to a single simulation. The separation of the simulation from the user interface allows the simulation to be run on a compute server,

¹This research was supported in part by the Department of Defense Advanced Research Project Agency, under contract N0039-84-C-0165, and by the New York State Science and Technology Foundation, under contract NYSSTF CAT (86)-5.

advantage. DL remote procedures, being simple, can be developed in collections much like local libraries are. A remote standard I/O library or a distributed math library, for instance, may be available. Distributed applications programming then is carried out mainly on the client process's machine and not on two machines. Distributing a program should not entail distributing the programming effort.

Conclusion

Three special purpose user-space protocols have been described:

- i. The TUBE - a protocol developed to provide a short term solution for file archiving on a machine short on file space and lacking a removable media capability.
- ii. Bulk Copy - a protocol developed for performance needs not being met by the general purpose protocol available. Bulk Copy is a stop and wait protocol which due to environmental circumstances performs as well as a sliding window protocol fraction of the implementation cost.
- iii. Distributed Library - a high level protocol built on a low level special purpose protocol. Special purpose protocol development doesn't necessarily preclude applicability to general purpose goals.

When a particular application under development requires a communication feature which is not available using commercially available standard products for the application's operational environment, the development of a special purpose protocol can be a reasonable approach. Environmental factors may lead the developer to unconventional approaches, which may be, nevertheless, appropriate.

References

- [Hardwick & Lekashman 1987] K. Hardwick and J. Lekashman, "Internet Protocol on Network Systems HYPERchannel Protocol Specification" RFC Draft, October, 1987.
- [Zwaenepoel 1985] Willy Zwaenepoel, "Protocols for Large Data Transfers over Local Networks", Rice COMP TR85-23, July, 1985.

basic features and usage of Nest.

The program in Figure 1 is a complete Nest program in C. It can be compiled and linked with the Nest library to create a Nest simulation program. The

first thing you may notice is the absence of a *main()* routine. While the programmer can supply a *main()* routine for the emulation program if desired, the Nest library contains a generic main routine which initializes the simulation with *node_main()* as the main routine for each simulated node. This main routine for each node takes a single argument, the node id assigned to it by Nest. Node ids are used by Nest to uniquely identify each node, and are used whenever a node needs to be specified for a Nest function.

An important part of a network simulation is communications between nodes. An example of this can be seen at the beginning of the *node_main()* routine. The first thing the routine does is to broadcast a message to all its neighbors. A message in Nest `#include <nest.h>`

```
#define HELLO 1
#define ACK 2

struct timeval five_seconds = { 5, 0 };

node_main (nodeid)
ident nodeid;
{
    char *message;
    ident dest, sender;
    int msgtype;

    broadcast (HELLO, "hello, world");          /* broadcast hello message */

    slumber (&five_seconds, SLUMBER_NOWAKE);    /* wait for hellos */

    while ( any_messages ( ) ) {                /* avoid blocking on recvm */

        sender = recvm (&dest, &msgtype, &message);

        hold ( );                               /* begin critical section */

        printf ("%d received\n",
                nodeid, message, sender,
                dest == 0 ? "broadcast" : "sendm" );

        release (1);                             /* end critical section */

        if (msgtype == HELLO)                   /* acknowledge hello message */
            sendm (sender, ACK, "isn't that a bit cliched?");
    }
}
```

consists of two parts, usually called the key and the data pointer. While these are both just 32 bit quantities, they are conventionally used in different ways. The key is typically used to identify messages, either by type or by number. The data pointer is usually a pointer to the data of the message, in some format determined by the type of the message. In the example, the key is the defined constant HELLO, and the data pointer is just a pointer to the null-terminated string "hello, world". This simple message structure of key and data pointer is extremely flexible, since the data pointer can point to any sort of data, from character strings to complex linked data structures, such as trees and lists.

After communication, the next most important thing in a simulation is the passage of time. Since all the processes in a simulation are running in a single Unix process, and since the passage of time within the simulation is largely independent of real time, system calls such as *sleep()* and *time()* will not have the

Figure 1

desired effect. Instead Nest provides alternate routines, such as *slumber()*, which is the next function called in the example. After a node broadcasts the hello message, it would like to receive messages from its neighbors. But it may take a certain amount of time for the messages to arrive. So *slumber* is called to suspend the node (in this case, for five seconds) to allow messages to arrive. The *SLUMBER_NOWAKE* parameter is a defined constant which tells Nest not to interrupt the *slumber* if messages arrive.

Once the node has waited a certain amount of time, it calls *any_messages()* to see if any messages are available to be received. This prevents the node from blocking indefinitely on a receive if there are no more messages which have been sent to it. While there are messages to be received, the node calls *recv()* to receive the message. It passes three pointers to variables, which are set to the original destination, key and data pointer of the message. The original destination stored in *dest* is just the node id of the receiving node, or 0 (which is not a valid node id) if the message was broadcast. The nodeid of the sender of the message is returned by *recv()*.

Once the message has been received, the example prints a diagnostic message describing the message which has been received. Before it does, it calls the *hold()* function to prevent Nest from interrupting the *printf* call and giving control to another node. This ensures that the messages from different nodes will not be mixed together, as well as preventing any problems caused by non-reentrant implementations of the *stdio* library. The *release()* function is called afterward, indicating that the critical region has ended. The parameter to *release* indicates the number of nested *hold()* calls to be released.

Finally, the node replies to each received HELLO message with an acknowledgment. Since the acknowledgment is directed to the node which sent us the HELLO, *sendm()* is called instead of *broadcast()*. The first parameter to *sendm* is a destination node id; otherwise it is identical to *broadcast*.

Implementation of Nest

Nest is implemented as a library that is linked together with the user's code. The user sets up the network and any other global data structures required, and then begins the simulation by calling the *simulate()* function. The simulation, once started, proceeds in a series of *passes*. During a pass, each node is allotted a fixed amount of simulated time called the *pass* time. Time may be consumed by running, sleeping, or waiting for the arrival of a message.

During a pass, the network remains fixed, so that all nodes run with a consistent view of the network. Any changes to the links or the nodes are made between passes. These changes can be made from a user interface client, or from a special *monitor function* which is called before each pass. The *nest_monitor()* function is supplied as a default—it

merely reports any changes (generally that a node has terminated) to the user interface clients, but does not alter the simulation otherwise. The simulation proceeds until all of the node functions terminate or are blocked waiting for messages (*i.e.* deadlocked).

The two most important functions of the Nest implementation are (1) performing the context switching between the nodes, and (2) ensuring that messages arrive in the proper order in simulated time. The context switching is done by setting a timer interrupt that goes off when the node has used up an amount of time equivalent to the pass length. In the case when the node does not perform any receive operations during its quantum, the timer goes off and the Nest timer interrupt handler routine gets control. It copies the registers, the global variable *errno*, and the stack below the *simulate()* routine into a save area using an assembly language routine. The next node is scheduled, and this information is restored (if it is being resumed) or initialized (if it is running for the first time).

If a *recv()* call is made, the node must be suspended. This is because other nodes which are earlier in simulated time might send a message that would arrive before the time of the *recv()* in simulated time. Even if there is a message in the queue at the time of the *recv()* call, another message might be sent later in real time which arrives earlier in simulation time. In order to ensure that messages are dequeued in the proper order, the node with the earliest simulated time is always run first. A node is suspended either because it is at the beginning of its quantum, or because it was blocked by issuing a *recv()*. In the latter case, when we resume the node we know that we can safely dequeue the earliest message in the queue because there are no other nodes with an earlier simulated time that could send a message.

In the previous section the *any_messages()* function was introduced. While conceptually simple, its implementation is fairly complex because of the necessity to maintain temporal consistency. It first checks if any messages are in the queue that have already arrived (that is, that have an arrival time less than or equal to the current time). If there are, it returns true. If not, it must invoke the scheduler just like *recv()*, and for exactly the same reasons: nodes with an earlier simulation time might send a message that will arrive before the current time.

Note that *any_messages()* does not use any simulated time—conceptually it always returns “right away”. However, *recv()* may or may not use simulated time: if a message arrives before the time of the *recv()* call in simulated time, the *recv()* will invoke the scheduler but when the node is resumed it will be at the same time; if a message does not arrive until later in simulated time, it will be resumed at a later simulated time, assuming that a message is eventually sent. We distinguish between these two states by calling the former *waiting* and the latter *blocking*. The

any_messages() function only waits, while *recv()* may do either.

Since one node function will generally be used on more than one node, the node functions must be re-entrant (that is, no global or static variables). The *hold()* and *release()* calls described above must be used to bracket non-reentrant code, such as accesses to global structures or calls to non-reentrant system routines. When using Nest, *malloc()* is redefined to call the *hold()* and *release()* routines, since *malloc()* is non-reentrant.

Advanced Uses

The existence of channel functions and the monitor function, combined with Nest's simple yet powerful network representation scheme, allows Nest to be used for a variety of purposes. Since the monitor function is called at the start of each pass, when the simulation is in a consistent state, it can be used to collect statistics at discrete intervals. The monitor function can also be used to dynamically alter the network topology under program control.

In addition, the channel functions can be written to model any kind of transmission behavior. The default function, *reliable()*, models the kind of connection provided by TCP, providing a fairly high-level abstraction of a network. However, channel functions can be written to model the network at a lower level, including such things as noisy lines, transmission delays, etc. Similarly, the monitor function can be written to model unreliable processors by "crashing" nodes.

In fact there is a stack of functions associated with each channel, and each function can filter the message as desired. This makes it easy to combine several different channel behaviors implemented by different functions, and to change the behavior of the channel dynamically.

The next section illustrates some of the uses for Nest by presenting a number of applications which have been developed using Nest. These range from a prototype of a complex distributed algorithm for position location to a detailed simulation of the ARPANET routing and topology update algorithms.

Applications of Nest

IPLS - A Distributed Incremental Position Location System

The first application to be developed using Nest was IPLS. The basic idea of IPLS is that nodes in a mobile packet radio network can determine their relative positions from communications delays. Distance information between nodes can be derived from fine measurements of the delays, and this distance information, combined with topological analysis of the network, is sufficient to recast the problem of relative position location as a problem in graph rigidity. Once the relative positions are fixed, absolute positions can be determined, given that the absolute positions of

three of the nodes are known.

However, since we had no packet radio network on which to develop a distributed implementation of IPLS, we developed Nest in order to give us a testbed in which to do this work. A number of features of Nest helped tremendously in developing IPLS. In particular, the ability to ignore all the lower layers of the network made the implementation of the basic protocol extremely simple. The ability to pass complex data structures as messages between nodes made the implementation of the second phase much easier, since the intermediate results could be passed "as-is", without serialization. Additionally, when the data structures used to implement the intermediate forms changed, which happened a number of times, none of the communications routines needed to be rewritten.

Since the raw data for the IPLS algorithm is the network topology itself, we needed to be able to test IPLS with a large number of network configurations. The ability of Nest to specify network configurations interactively, via the user interface, made this task much easier.

Simulation of Microeconomic Model for Load-Balancing

Nest has also been used to develop a simulation of a load-balancing system based on microeconomic principles. The basic idea of the work is that the laws of supply and demand could be used to equitably distribute CPU and communications loads among a number of processors. Each process which has work to do also has a certain amount of "money", which is used to bid for CPU and communications resources. If a processor is overloaded, while neighbors are underutilized, bidding will force up the "price" of CPU time at that processor, and processes will decide to migrate themselves to neighbors where the price of CPU is lower.

A number of alternatives were explored in deciding what system would be used to implement this simulation, and there were several reasons why Nest was chosen over more traditional discrete-event type simulations. The microeconomic model presumes a certain amount of basic intelligence in the processes themselves, since it is they who decide when and whether they will migrate from one processor to another. With Nest, it was easy to write a number of C functions which implemented various degrees of intelligence for the processes; these were simply called by each node's main routine when appropriate. Additionally, the somewhat complex bidding procedure used on each machine to determine pricing and allocation of resources was quite naturally built in to the main routines for each node. In a discrete-event based simulation package these sorts of behaviors would have been more awkward to simulate.

A Study of the Topology Update Problem in the ARPANET

Another project which used Nest was a study of the ARPANET topology update problem. The topology update problem is a phenomenon which exists in store and forward networks with dynamic routing of packets. Since each node maintains a local database of network connectivity and delays, it is possible for inconsistencies between nodes to cause problems such as looping and forms of lock-up.

This project used Nest to build a detailed model of a small ARPANET-type network, with hosts and end-node and intermediate IMPs connecting them. Both normal (data) messages and topology update messages were simulated. Extensive use was made of Nest's ability to run a user-supplied function at the start of each phase of the simulation. This monitor function was used both to collect statistics from the previous simulation phase, and to generate dynamic events, such as crashes and reboots of hosts and IMPs. The ability to dynamically alter network topology and crash and restart nodes in Nest was extremely useful in this project.

Other work and availability of Nest

Nest has been distributed to a large number of other sites. One site which has done extensive work with Nest is the Northrop Research and Technology Center, where Nest is being used to simulate some of the Noanet protocols. The Noanet is a network architecture based on flooding protocols. More details can be found in [Rose87].

The current release of Nest version 2.4 runs on Dec VAX minicomputers and 68000 workstations including Sun and Integrated Solutions 68K, under any 4.2bsd Unix derivative (including 4.3). A port to System V should be available shortly. The user interface client runs on Suns with 2.0 or later release of SunView. An X client developed at Northrop exists, but it depends on their own modifications to Nest. For details on obtaining Nest, contact the authors at the electronic or postal addresses above.

Conclusions

Results

As the previous examples have shown, Nest is a useful tool for a variety of network simulation problems. There are a number of reasons for its success as a network simulation tool. The major reason is that Nest has a very general, and programmable, model of network architecture. The division of Nest into a simulation server and user interface clients is another important reason.

Since Nest doesn't assume a specific network architecture or modeling method for the network, it can be used not only for studying problems using standard queuing models, but also for problems where other considerations are more important, or other approaches provide more useful results. The fact that Nest is a library designed to be used with a standard

language (in this case C, although interfacing to Fortran or Pascal would be possible), rather than a simulation package with limited programming capabilities, gives the simulation builder the ability to fine-tune the behavior of the simulation program, and the interaction between simulation objects.

The concurrent processing provided by Nest allows a more natural approach to the simulation of distributed systems, without the need for explicit coroutine structures. The provision of a monitor function which is run at the start of each phase of the simulation allows centralized functionality for statistics and logging, or code which deals with global data or file descriptors.

The simple communications facilities provided by Nest allow the user to be concerned with only the the highest (application) level of a protocol or distributed system, without worrying about lower-level issues. On the other hand, the user can also customize the behavior of these communications facilities to reflect the realities at any of the lower layers. The combination of these features with the ability to run ordinary C code in a concurrent processing environment makes Nest ideal for prototyping distributed systems.

The division of Nest into simulation server and user interface clients has proved to be a useful technique. It allows the simulation to be run on a large machine with more CPU resources, while the user interface can run on a workstation. More importantly, it separates the functions of computation and user interaction in a way that simply using a window server system such as X would not achieve, since Nest can have multiple user interfaces running at once, and users can connect and disconnect from the simulation at any time. This is especially useful for long-running simulation programs.

Future Directions

Our project group has taken Nest as a starting point for the development of a system for network monitoring and management called Net-MATE. This work takes the basic Nest components of simulations and user interfaces, and extends the structure to include not only multiple user interfaces, but also multiple sources of network information and control (for both real and simulated networks), all coordinated through a central network model database.

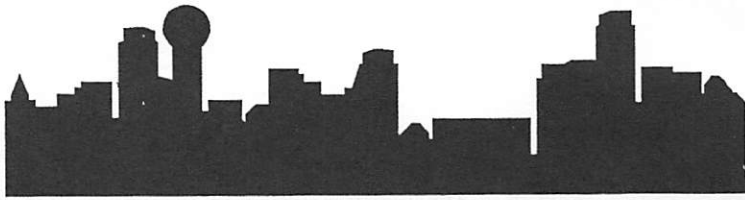
Unusual features of this project will include support for tracking multiple layers of network protocols, from physical to application, as well as support for user interfaces which deal with these multiple protocol layers in large and complex networks of hundreds to thousands of nodes.

There are also a number of areas more directly related to Nest in which future work could be done. One possibility would be to integrate Nest with an object oriented language, such as C++ or Objective-C, in order to support simulation programs with much of the flavor of Simula. The concurrent nature

of Nest implies that substantial speedups could be achieved by a parallel implementation for shared-memory multiprocessors. While this would require a major overhaul of Nest, the resulting performance increases could well be worth it.

References

- [Bacl87] Kenneth Baclawski, *A Network Emulation Tool*, Proceedings of the Symposium on the Simulation of Computer Networks, 1987.
- [Cook87] Robert P. Cook, *Starlite, A Network-Software, Prototyping Environment*, Proceedings of the Symposium on the Simulation of Computer Networks, 1987.
- [Kepe86] Jonathan Kepecs, *Lightweight Processes for UNIX Implementation and Applications*, Proceedings of the Summer Usenix Conference, 1985.
- [Libe87] Don Libes, *Multiple Programs in One UNIX Process*, Usenix Association Newsletter, Volume 12, No. 4, 1987.
- [Rose87] Marshall T. Rose, "The Nest Simulation Testbed at NRTC", Northrop Research and Technology Center Technical Paper, 1987.
- [Xu87] Chong-wei Xu, *A Simulation Test Bed for Computer Networks*, Proceedings of the Symposium on the Simulation of Computer Networks, 1987.



V. S. Sunderam
Dept. of Math & Computer Science
Emory University
Atlanta, GA 30322
gatech!emory!vss

A Fast Transaction Oriented Protocol for Distributed Applications

ABSTRACT

Distributed computing environments consisting of independent processors connected by high-speed LANs are becoming increasingly common, as opposed to processors under control of a distributed operating system. However, the classes of distributed applications possible and in existence in such environments are limited – primarily by the communications transport mechanisms available. Traditionally supported protocols do not provide the functionality and performance required for potentially viable, novel uses. In this paper, the design of a simple but fast transport protocol to provide transaction oriented services over a LAN is presented. The protocol is message oriented, uses a global addressing scheme, and incurs minimal processing and transmission delays. Although not mandatory, a procedure-call interface between the application and the transport provider is implied in the protocol design; enabling straightforward implementation and efficient operation. An implementation under SunOS and preliminary results are discussed and comparisons against existing protocols presented.

Introduction

A large class of distributed applications may be built in computing environments consisting of independent processors connected by high-speed local area networks. The facilities commonly available permit these applications; the sophistication and coherence of a distributed operating system are often not required to support them. However, the functional and performance requirements of some of these applications and other potential ones are often thwarted – primarily by the limitations of the end-to-end communication facilities provided. These facilities, while well established, well understood, and stable, are overly general and therefore complex and expensive. This paper presents preliminary results in the design and use of a simple, lightweight, protocol aimed at providing a fast, transaction oriented, transport mechanism on a local network.

Networks of Unix workstations are commonplace and several distributed applications have been implemented in such environments. Representative examples are network file systems, remote procedure calls, and remote execution of programs. These and other applications utilize Internet protocols on most Unix systems; specifically, UDP or TCP is the transport protocol of choice. These applications perform adequately but their use of Internet protocols is not without drawbacks. For instance, to achieve acceptable performance, the SunOS implementation of NFS

necessitated substantial changes to the existing UDP/IP implementation in the kernel[1]. The SunOS RPC facility using UDP is unreliable, using TCP (or stream RPC) is expensive for a single request-reply transaction. Remote execution with `rsh` again uses stream connections, unwarranted if it is only required to execute a single, small, command on a remote machine.

The need for a fast, reliable, message oriented transport mechanism already exists. Furthermore, if such a mechanism were available, other applications may become viable. One example is distributed debugging, where a local process may wish to set breakpoints, examine data values or otherwise control the execution of a remote process. Another example is process migration – an application that requires rapid and reliable transfer of a process core image. Real-time load sampling to determine the best processor on which to execute a short command is yet another possibility. Also viable is a network based library server – useful for exploiting specialized hardware or software available only on a particular processor. In these and other cases, the sophistication provided by TCP is not required and the connection establishment overhead can be prohibitive; on the other hand, the inherent unreliability of UDP is unacceptable. Moreover, a large percentage of applications are constrained to a single, local network and therefore need not incur the overheads of generalized Internet

addressing and routing.

The primary obstacle in the development of more appropriate network software is complexity – of the protocol design as well as its implementation. The most critical issues are siting (the positioning of the protocol relative to the kernel and user processes), modularization (the mapping between protocol layers and software modules), and demultiplexing (the scheme used by the protocol provider to support multiple applications). Detailed analyses of these issues may be found in [2] and [3]. In the latter paper, however, a mechanism termed the *packet filter* is described, which provides user-level processes with efficient access to the network interface. The packet filter has evolved over several years and implementations for various Unix systems have been reported. SunOS supports a primitive version of the filter termed the *network interface tap*; it is reasonable to expect that Unix systems will continue to provide such a facility in future.

In this paper, a transaction oriented protocol that can be built on such a network interface access point is described. This protocol, termed TOP, is intended to provide reliable message transfer between two hosts on a local network with minimal delay and overheads. It is designed for applications that follow the client-server model and require rapid but possibly sporadic transfer of data units of varying sizes. The key features of the protocol include application-controlled reliability levels and a design biased toward a procedure call interface between the application and the protocol provider. The protocol is described in the next section following which preliminary implementation results and comparative measurements are presented. A discussion of ongoing and future work concludes the paper.

The Protocol

The TOP or Transaction Oriented Protocol is defined to make available a message transfer mode of transport available on a local network. It should be noted at the outset that the design and semantics of TOP have been influenced by the intended Unix 4.xBSD environment; however, implementation in other environments should be straightforward and as effective. The primary purpose of TOP is to enable a Unix process on one processor to transmit a *request* message to a process on a different processor and to receive, in response, a *reply* message. Communication using TOP is asymmetric and follows the client-server model; the protocol does not require the server to be able initiate a transaction. TOP is not connection oriented in the interest of simplicity and efficiency; on the other hand, the protocol guarantees delivery and protects against duplication. Sequenced delivery is not explicit in the definition but is expected to be inherent in the implementation. In addition, options may be specified on each transaction that may enable the implementation to provide differing qualities of service. The protocol does not impose constraints on the

size of messages and should therefore provide for fragmentation and reassembly as required. TOP supports several of the features found in more sophisticated protocols such as VMTP [4] and those suggested in [5]. However, it should be noted that the protocol is still evolving and does not as yet incorporate mechanisms for authentication, forwarding, etc.

Application Interface

The TOP protocol is intended to provide Unix processes with facilities complementary to those already available – specifically a fast and reliable message delivery mechanism. It is also intended to be simple and straightforward both to implement and to use. Given the complexity of kernel implementations and the overheads in context switching between processes under Unix, it is suggested that the protocol-application interface be procedure call based (the remainder of this paper assumes such an interface). The model therefore is essentially RPC-like; on the client side the application makes procedure calls to the protocol, and on the server side, the protocol makes procedure calls on application routines. However, there are some significant differences and added functionality that are discussed in the following sections.

The client application communicates with the protocol by issuing a **top_request** and receiving a response, both contained in a single procedure call. The arguments of the call are explained below; the return value of the request is set to be that returned by the remote application procedure. The notation $\langle m:n \rangle$ indicates a bit field beginning at position m for n bits; bit 0 is the least significant position.

top_request(*serv_mc*, *serv_proc*, *tos*, *msgout*, *msgoutlen*, *msgin*, *msginlen*);

- *serv_mc*, *serv_proc*: Strings specifying the host-name of the server processor and the name of the server procedure providing this service. (The next section describes naming and addressing in more detail).
- *tos*: Type or quality of service desired. *tos* is a 16-bit value-result argument and contains the following information in the request.
 - $\text{tos} \langle 15:3 \rangle = \langle \text{Unused} \rangle$
 - $\text{tos} \langle 12:1 \rangle = \text{Unreliable service acceptable}$. This flag may be used in circumstances where the application does not need a high level of reliability or when the application provides its own end-to-end reliability. Normally, the implementation will be able to achieve greater throughput if the application specifies this option.
 - $\text{tos} \langle 11:1 \rangle = \text{Time the transaction}$. This option causes the protocol to return a measure of the elapsed time to complete the transaction and may be used for statistics gathering or monitoring.

- *tos<10:1>* = Deliver once. With the once-only option, applications can ensure that a message is delivered exactly once. Even though retransmissions may be necessary, the application server will receive the message only once. However, enabling this option may result in the loss of the response message.
- *tos<9:1>* = No reply. No response message is expected; delivery of the request message is guaranteed.
- *tos<8:1>* = Checksum. The protocol will perform checksumming to ensure data integrity in addition to guaranteed delivery if this option is used.
- *tos<7:8>* = Timeout value. This field allows the application to specify a timeout value in seconds that will be used by the protocol to control retransmissions. A zero value causes the default period to be used.
- *msgout, msgoutlen*: A pointer to the request message and an integer indicating the length, in bytes, of the message.
- *msgin, msginlen*: A pointer to a buffer to receive the response message and a value result parameter that holds the length of the incoming message, in bytes, upon return. The buffer should be large enough to accommodate the incoming message..

The **top_request** procedure call returns with the *msgin* parameter containing the reply message, the *msginlen* parameter containing its length and the *tos* parameter set as follows:

- *tos<15:4>* = *<Unused>*
- *tos<11:1>* = Failure. A 1 in this bit position indicates protocol failure; the message may have been delivered zero or more times.
- *tos<10:1>* = Delivered once, reply lost. The message was delivered exactly once (as specified in the request) but the response message could not be delivered.
- *tos<9:1>* = Service unavailable. The TOP server on the specified processor is not operational.
- *tos<8:1>* = Non-existent service. The (*serv_mc*, *serv_proc*) pair that the reply was addressed to does not exist.
(Note: Zeroes in bits *<15:8>* indicate successful delivery and response.)
- *tos<7:8>* = Transaction time. If a timing request was made, this field contains the actual time for the complete transaction.

On the server side, the protocol makes a procedure call on the particular server procedure as follows:

serv_proc(*in, inlen, out, outlen*);

- *in*: A pointer to the message received from the client.
- *inlen*: An integer value representing the size of the incoming message in bytes.
- *out*: A pointer to a buffer in which the return message is to be placed. The size of the buffer is implementation dependent; a default of 16K bytes is recommended.
- *outlen*: A value-result parameter in which the application procedure must place the size of the response message, in bytes.

Addressing

The addressing mechanism used in TOP is still primitive at this stage, but has been found to be sufficient for several applications. A server procedure is specified by the pair (*serv_mc*, *serv_proc*), both of which are string valued, representing the hostname and procedure name respectively. Since it is expected that application procedures will be bound into the TOP server, this necessitates unique procedure names (at least per server). Depending on the lower level communication mechanism used, multiple servers may exist on a host; our preliminary implementation consists only of one server on each host into which all application procedures are linked. A simple front end to **ld** is made available that allows the addition or deletion of user procedures to a server executable file. At present, multiple versions of the same procedure are not supported.

The server interface to application procedures therefore consists of a simple switch to the appropriate procedure for a given request – the **ld** front-end program modifies the switch as necessary when application procedures are added or deleted. A list of server procedures and the hosts on which they are available should be publicly available. This list is also maintained by the front-end and, in the experimental implementation, is stored in a network-wide file which is read in by clients. Address processing in the **top_request** call therefore involves validating hostnames and procedure names against static tables, the mapping of hostnames to local network addresses, and of procedure names to unique integer identifiers (purely for implementation ease).

While it is crude and not without drawbacks, this simple, ad-hoc addressing scheme has been found to work well. In particular, the avoidance of port numbers, and static translation into network addresses contributes significantly in reducing overheads. In addition to the above, clients may also specify “any”, “all”, or “multicast” as the hostname to which the request is to be sent. In the first case, the TOP provider selects an arbitrary server that provides the service; in the second, the message is transmitted to all servers that provide the requested service with the first response being returned to the client. The “multicast” facility returns all responses

received from an “all” transmission with the responses are concatenated in the order they are received. At present, applications need to incorporate mechanisms to identify the source of responses when using these special addressing facilities.

The Lower Interface

As mentioned, the TOP protocol is intended to be built directly over the network interface although it is possible to use a different lower layer – *e.g.* raw IP. The only requirement of the lower layer is that it be possible to transmit an arbitrary packet (possibly of limited size) to another host on the local network; the TOP protocol should be able to supply the data-link header if this is necessary. The packet should be identifiable based on at least one field in the data-link header or in the remainder of the packet – for packet reception, the lower layer should deliver packets distinguished by this identifier and addressed to the current host. The 4.3BSD *packet filter* and SunOS *nit* provide this functionality; the following discussion is in terms of the *nit* interface.

The *nit* interface provides a tee connection to a specified network interface [6] and specifically, on Ethernet, permits unprocessed packet read and write capability. On SunOS, a *nit* socket may be created and bound to an interface by the calls

```
s=socket(AF_NIT,SOCK_RAW,NIT_PROTORAW);
bind(s,<sockaddr structure with interface
name>,<size>);
```

The “operating modes” including filtering criteria, minimum blocking factors, and flags are set using *ioctl* system calls – in particular, only packets with a specific Ethernet type field may be requested. Incoming packets are returned by *read* system calls and include both the Ethernet header and the *nit* header. Packet transmission is done using *sendto* system calls with a *sockaddr* structure containing the destination Ethernet address and the AF_UNSPEC address family.

Protocol Format & Operation

This subsection describes the rules of procedure and header formats used by the TOP protocol. The structure of the TOP header that is prepended to each

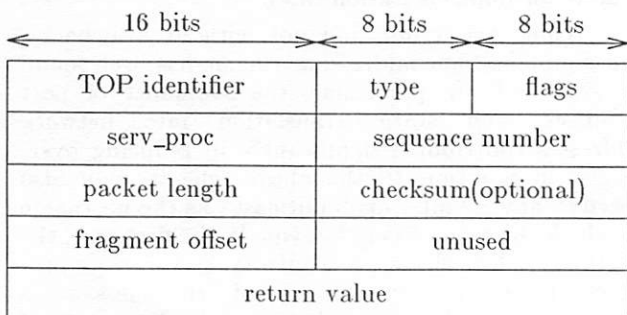


Figure 1: TOP header format

transmitted packet is shown in Figure 1.

The ‘TOP identifier’ field is provided to enable servers to uniquely identify multiple clients on a client host. The most natural value for this field is the client’s process id. At present, only two types of packets are used – the ‘type’ field therefore either contains ‘1’ or ‘2’ indicating requests and responses. The ‘flags’ field is used by both the client and server and contains the following information:

- *flags*<7:1> = Checksum required. If this bit is on, the ‘checksum’ field contains a 16-bit checksum of the entire message. Checksums are used either on user request or automatically by the protocol if the message is fragmented. In the former case, the server must also checksum the reply message.
- *flags*<6:1> = More fragments to follow. If a message is fragmented, this bit is turned on and the ‘fragment offset’ field contains the byte offset of this packet in the original message. The last fragment is identified by a non-zero value in the ‘fragment offset’ field. Both the client and server may fragment messages.
- *flags*<5:1> = No acknowledgment. This bit is turned on by the client if unreliable data transfer is acceptable. Servers do not acknowledge messages; it is assumed that applications perform their own end-to-end verification.
- *flags*<4:1> = No reply/Service unavailable. Clients do not expect a reply message from the server – this facility permits servers to acknowledge messages before handing them to application procedures, thereby minimizing latency. The same flag is used in server replies to indicate a non-existent server procedure.
- *flags*<3:1> = Deliver once. This flag is used by clients to indicate ‘once only’ delivery requirements.
- *flags*<2:1> = Negative acknowledgment. Servers that detect an error when reassembling fragments respond with this flag.
- *flags*<1:1> = Previously acknowledged. Servers use this flag to indicate the reception of a duplicated ‘once only’ request from a client that was previously delivered to the server procedure and acknowledged. Subsequent requests are not delivered to server procedures.
- *flags*<0:1> = Normal request or response.

The ‘sequence number’ field contains a modulo 2^{16} sequence number that is used to verify acknowledgments as well as to implement ‘once only’ delivery. The ‘packet length’ field contains the length in bytes of the current packet including the TOP header. At present, string valued procedure names are mapped into integers for ease of implementation, this id is contained in the ‘serv_proc’ field. The ‘return value’ field holds a value, if any, returned by the application procedure – interpretation of its type is

application dependent.

Client Operation

The TOP provider on the client receives a message transmission request as described in section 2.1. The algorithm executed by the client is extremely simple. Retransmissions are controlled either by timeouts or checksum failures but are only initiated by the client. Acknowledgments from the server are implicit in the response message; lost requests and replies are detected only by the above mechanism. As a procedure call interface is being assumed, a request is completely processed before the next is accepted – the intended use of TOP suggests that implementations should avoid servicing concurrent requests. The steps performed are listed below.

- (a) Check validity of the server procedure to which the message is to be delivered. (The mechanism used is implementation dependent). If invalid, return 0 to the requestor with *tos* < 8:1 > set.
- (b) Determine the number of fragments necessary for this message. A fragment should preferably be as large as the data-link layer will allow. If checksumming has been requested or the message is to be fragmented, compute a 16-bit checksum on the entire message.
- (c) For each fragment, compute the TOP header fields and transmit the packet. Depending on the data-link interface, an appropriate data-link header may have to be constructed.
- (d) Set a timeout using the user-supplied value or the default value. On the occurrence of a timeout, repeat step (c). If too many (default 5, implementation dependent) timeouts have occurred, return 0 to the requestor with *tos* < 11:1 > set.
- (e) Receive packets from the data-link layer. Discard any packets that do not contain the same 'identifier' and 'sequence number' fields as in transmitted packets. If the response message is fragmented, reassemble packets as they are received. If the 'previously delivered' flag is set in the incoming packet, return 0 to the requestor with *tos* < 10:1 > set.
- (f) On receipt of the complete message, compute a 16-bit checksum if requested by the application or if the reply message was fragmented. Compare this checksum against that in the TOP header and return to step (c) if not equal.
- (g) Return to the application with the value from the 'return value' field and the incoming message and its length in the appropriate parameters. The lower order byte of *tos* should contain the actual time in seconds for the complete transaction if this was requested.

Notes: In the case of a fragmented message, the checksum is carried in the headers of all packets. Fragments may be received in any order but the receipt of the last logical fragment terminates the

reassembly of the message.

Server Operation

Depending on the demultiplexing and filtering options provided by the data-link layer, more than one server process may exist on a host processor. For instance, SunOS *nil* permits filtering on Ethernet packet type; multiple servers could be based on different values in this field. Assuming that application procedures are bound into the server executable file, the TOP server will normally remain in an idle state, awaiting an incoming request. On receipt of a packet, the server "locks in" to packets with the same 'identifier' and 'sequence number' fields until all constituent fragments of the current message have been received. The checksum (if appropriate) is then computed, the message is discarded in the case of a mismatch and a 'Negative Acknowledgment' is transmitted to the client. Again, a complete message is processed before accepting the next; intervening requests from other clients are therefore discarded and serviced when they retransmit.

Once a request has been assembled, the server procedure to which the message is addressed is invoked. If no such procedure exists, a 'Service Unavailable' response is transmitted back to the client. On return, the response message is transmitted to the client, in fragments if necessary, with the 'return value' field in the TOP header containing that returned by the application procedure. The response message is the implicit acknowledgment for the incoming request – therefore, the 'identifier' and 'sequence number' fields are copied from the incoming header. The server assumes that the data-link layer provides information concerning the originating host. On *nil*, the Ethernet header containing the source host's Ethernet address is made available with the incoming packet.

The server also maintains the sequence number of the last message successfully received and acknowledged for each client in order to implement 'once only' delivery. Incoming requests that specify 'Unreliable delivery' are not acknowledged and servers checksum the response either if the incoming request was checksummed or if the reply message is fragmented. Server responses are not re-acknowledged by clients and lost replies are handled by client timeouts and retransmission.

Pragmatic Considerations

As previously mentioned, several aspects of the protocol design are influenced by the assumption of a procedure call interface both on the client and the server. In particular, the exclusion of transactions once processing has begun on a message may be limiting under some circumstances. However, given that intended applications will typically complete a transaction in a few hundred milliseconds at most, the simplicity and efficiency gained argues in favor of this approach.

The TOP protocol does not explicitly address such issues as network byte ordering or data type representation. This is deliberate; in several instances clients and servers will, in fact, execute on identical processors. If required, the XDR [7] standard may be very easily incorporated into the TOP interface to support machine-independent data representation.

TOP does not provide an explicit mechanism for flow control. It is therefore possible, in the case of very large messages, that the client transmits fragments at a faster rate than the server can receive and reassemble them. During testing of example

implementations, it was never possible to induce this situation; nevertheless, a simple flow control mechanism is being designed to preempt such circumstances. Overruns may also occur when clients rapidly transmit messages using the 'Unreliable' mode of operation. It is not clear whether flow control mechanisms should be used to reduce packet loss in this situation.

Both the feasibility and the effectiveness of providing a TOP protocol service depends considerably on the ability to by-pass the traditional protocol hierarchy and directly access the local network

Packet size (bytes)	RPC/UDP Spray			Unreliable TOP Spray			Reliable TOP Spray	
	Pkts/sec	KB/sec	% Loss	Pkts/sec	KB/sec	% Loss	Pkts/sec	KB/sec
86	531	45	2.4	581	49	1.3	219	19
502	373	187	3.8	468	235	2.1	158	79
1002	281	281	4.8	326	326	2.5	129	129

Figure 2: Unreliable Data Transfer

Block size (bytes)	Using TOP seconds to transfer:			Using TCP seconds to transfer:		
	100KB	500KB	1MB	100KB	500KB	1MB
1K	0.78	3.70	7.11	0.86	2.23	3.88
4K	0.71	2.81	5.43	0.85	2.36	4.33
8K	0.63	2.73	4.93	0.78	2.55	4.82

Figure 3: Reliable Data Transfer Times

Block size (bytes)	TOP transfer(%loss) seconds to transfer:			UDP transfer(%loss) seconds to transfer:		
	100KB	500KB	1MB	100KB	500KB	1MB
1K	0.70(0)	1.5(0.2)	2.15(0.8)	0.78(0)	1.8(0.8)	2.41(4.3)
4K	0.56(0)	1.8(0.1)	2.7(2.4)	0.63(0)	1.24(2.5)	1.90(2.5)
8K	0.48(0)	1.7(0)	3.2(1.6)	0.58(0)	1.0(5.2)	1.78(9.9)

Figure 4: Unreliable Data Transfer Times

Message size (bytes)	TCP transaction time (msecs)	UDP transaction time (msecs)	TOP transaction time (msecs)
100	16.6	6.6	5.9
200	16.6	8.2	6.3
500	19.6	9.8	7.1
1000	23.0	11.1	8.6
2000	37.8	15.3	14.1
4000	46.0	25.1	22.1

Figure 5: Single Transaction Times

interface at the data-link level. It is appreciated that this may always not be possible or only under restricted circumstances. As much of the benefits of TOP are expected to result from the ability to transmit and receive unprocessed or minimally processed packets, it may not be worthwhile to implement above, for example, raw IP.

Implementation and Experiences

The design of the TOP protocol was done concurrently with a preliminary implementation with the objectives of (a) permitting partial design modifications to the protocol if this could benefit the implementation, and (b) demonstrating that a straightforward implementation with minimal complexity and effort was possible. At present a Sun-3 implementation under SunOS exists and a 4.3BSD implementation for a Vax-11/780 is under development. Comparative measurements of performance and reliability have been made against TCP and UDP and are reported here; also discussed are novel applications under development.

SunOS Implementation

The TOP protocol has been implemented on SunOS 3.4 for Sun-3 workstations and uses the *nit* interface mentioned earlier. Implementing TOP was a straightforward process and the software consists of two halves – a set of library routines containing the **top_request** user interface for use by clients and a loader front-end program that accepts user procedures and links them to a standard TOP server program. The **ld** front-end program maintains a global services file and protects against name conflicts; it also allows the deletion and replacement of application server procedures.

For efficiency as well as to avoid preempting the use of signals by client applications, the retransmission mechanism in the client library is implemented using the timeout facility in the **select** system call. The basic actions in the **top_request** procedure consist of performing validation checks followed by transmission of the message and waiting for an acknowledgment piggy-backed on the reply message. Individual fragments are not acknowledged; instead, a checksum is forced on fragmented messages as a means of detecting lost fragments. On checksum failure, the server issues a negative acknowledgment resulting in the entire message being retransmitted.

The server is normally blocked on a *nit* socket read; on receipt of a message, basic validation checks are performed. The appropriate application procedure is then invoked and the reply transmitted back to the client. A record is maintained of the last successfully received message for each client in order to implement 'once only' delivery. No retransmission mechanism is used by the server. The TOP implementation consists of approximately 800 lines of C code and was developed over 2-3 weeks.

Results

In order to measure the performance and effectiveness of the TOP protocol, comparisons were made between applications using TOP and identical programs using TCP/UDP. The results of these measurements are reported in this subsection; in each case, the same experiment was repeated several times and the and the observed figures averaged. It should be noted that the figures shown are for a first-cut, unoptimized, TOP implementation.

Unreliable Transfer Rate

The Unix **spray** program uses UDP based RPC to measure transfer rates between two hosts. The **spray** program was rewritten using TOP and measurements were made under both reliable and unreliable modes of operation. Figure 2 shows the transfer rate between two Sun 3/60's using UDP based and TOP based versions of the **spray** program; 1000 packets were transmitted in each case.

For all packet sizes, the unreliable TOP version was found to provide between 9% and 25% more throughput while reducing the number of lost packets by approximately 50%. In the reliable version of the TOP **spray** program, the transfer rates were found to be slightly less than half of that observed with RPC **spray**. This is consistent with the round-trip time for each acknowledged packet and the added network latency.

Bulk Data Transfer

Although it is not intended that TOP be used for bulk data transfer, experiments show that TOP compares favorably with kernel based implementations of TCP and UDP, for reliable and unreliable transfers respectively. Figure 3 shows elapsed times in seconds for reliable memory-to-memory transfers between two Sun 3/60's using TOP and TCP for varying amounts of data as well as with differing block sizes. It can be seen that for transfers up to sizes of 100K, TOP is approximately 10% faster than TCP owing to the fact that connection set up overheads are avoided. For larger transfers TCP outperforms TOP, particularly when using smaller block sizes.

Unreliable transfers of bulk data resulted in a lower percentage of loss when using TOP as shown in Figure 4. Owing to the fact that checksumming is done in TOP for fragmented messages even in the unreliable transfer mode, the TOP transfer rates are slower in some instances.

Single Request-Response

As the TOP protocol is primarily intended to support transaction oriented communication, the round-trip time for a single (perhaps isolated) request message and the corresponding reply is its crucial performance measure. Shown in Figure 5 are timings in milliseconds for single request-response transactions for varying message sizes, comparing the use of TCP,

UDP and TOP. In all cases, the request and response messages are of equal length and no processing is done by the server. The measurements are between a pair of Sun-3/60's with load averages slightly less than 1.0, and a moderately loaded Ethernet.

As expected, the TCP based transaction times are considerably large, indicating the connection set up overhead. The TOP timings show a 10-30% improvement over UDP; it should also be noted that the TOP transactions are reliable whereas the UDP transactions are not.

New Applications

Given the availability of a fast and reliable transaction oriented transport service, non-traditional distributed applications become viable. The feasibility and benefits of some novel applications using TOP are currently being investigated and preliminary findings are presented below.

Remote Execution

Remote execution programs such as `rsh` are used under a variety of circumstances – one of them being to execute a program on a lightly loaded processor in a network. Programs that locate a host with a low load average and execute on that host are common and software systems (e.g. [8]) that provide such facilities are also gaining in popularity. Sometimes, however, the statistics used in locating a lightly loaded processor are not sufficiently current e.g. those maintained by the Unix programs `rwho` and `rwhod`. Furthermore, in several cases, it is not necessary to start up a remote shell for the purposes of executing the program.

As a test application, a real-time load determination and remote execution program has been written that uses the TOP protocol for both determining a suitable remote host as well as for initiating the execution. The program presently supports user programs that perform only file I/O with the restriction that absolute pathnames that are valid on all hosts must be used. When a program is to be executed using this facility, the user types the command

```
remote <a.out>
```

The `remote` program is a TOP client that polls all servers that support remote execution. Each server returns real-time load information read out of the server host's kernel virtual memory. The least loaded processor is chosen and the client makes an execution request to the server on that host. The server forks a subprocess which executes the user's `a.out` file. Using a pool of six processors, it was found that the load determination phase requires between 50 and 65 milliseconds, including the necessary computations. For programs that require a few seconds to execute, the `remote` facility was found extremely useful and significant gains in overall program execution time as compared to local execution were observed. Enhancements are being made to this scheme to allow

interactive programs as well as to export environment variables to the remote host.

Network Shared Libraries

User programs normally link to several library procedures which, depending on the application, may be substantially large or computation intensive. To avoid duplicated copies of libraries in executable files, both on secondary storage as well as during execution, shared libraries are used in several operating systems. System V Unix supports shared libraries and the facility is being implemented in SunOS.

An interesting extension of this concept is to share libraries across a local network. Such network shared libraries may be valuable in certain circumstances e.g. when the memory requirements and computation costs of the libraries considerably outweigh the communications overheads. Libraries on selected hosts on a network may also be useful when only certain hosts possess specialized hardware or library software. In order to implement such a scheme, a fundamental requirement is a fast and reliable communication mechanism. The TOP protocol possesses the advantages of speed and reliability over RPC; further, features such as 'once only' delivery will be valuable in such applications. Potential library suites for experimenting with this concept are being evaluated; a strong possibility is the X window system which contains a large set of library procedures, and already uses asynchronous communication between clients and the display server.

Conclusions and Ongoing Work

The TOP protocol was designed and implemented to determine the feasibility of providing a fast, transaction oriented transport service on a local network. Initial testing has demonstrated that TOP, implemented completely at the user level, is capable of supporting transaction oriented communication at performance levels superior to that of existing protocols. The reliability of TOP in combination with its speed makes it attractive for both conventional and novel distributed applications. However, several aspects require refinement; work is in progress to improve and enhance features of TOP as well as to experiment with new applications.

Analysis and profiling has shown that much of the minimal transaction time of 6 milliseconds is spent executing Unix system calls. In an attempt to reduce this overhead, schemes such as that discussed in the Synthesis system [9] are being examined. Certain optimizations in the TOP client and server programs are also being implemented. The most significant deficiencies in TOP are felt to be the lack of a flow control mechanism and the primitive addressing scheme. The latter aspect has not been restrictive but may become so when a large number of applications are developed. A simple flow control scheme is being investigated where clients impose a self-adjusting delay between rapid sequences of

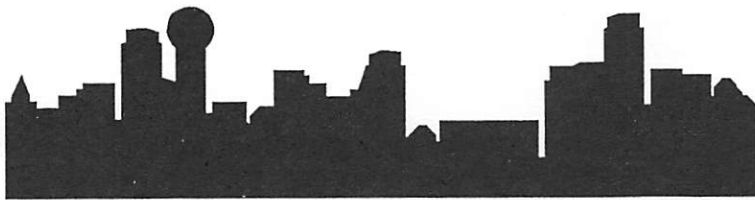
fragments. In addition to the library server application, the remote execution program is undergoing improvements and enhancements. TOP is also being incorporated into the process migration facility that is in use at the author's installation and extensions to **dbx** to enable remote debugging using TOP are under investigation.

References

- [1] R. Sandberg, "The Sun Network Filesystem: Design, Implementation and Experience", *NFS Tutorial Notes*, USENIX Winter 1987.
- [2] D. Clark, "Modularity and Efficiency in Protocol Implementation", *RFC817*, Information Sciences Institute, July 1982.
- [3] J. Mogul, R. Rashid, M. Accetta, "The Packet Filter: An Efficient Mechanism for User-level Network Code", *ACM Operating Systems Review*, Vol. 21, No. 5, November 1987.
- [4] D. Cheriton, "VMTP: Versatile Message Transaction Protocol", *Stanford University, Computer Science Department*, Preliminary Version 0.3, January 1987.
- [5] R. Braden, "Towards a Transport Service for Transaction Processing Applications", *RFC955*, Information Sciences Institute, September 1985.
- [6] Sun Microsystems, Inc., "Unix Interface Reference Manual", Sun Microsystems, Inc., 1986.
- [7] Sun Microsystems, Inc., "XDR: External Data representation Standard", *RFC1014*, Information Sciences Institute, June 1987.
- [8] D. Nichols, "Using Idle Workstations in a Shared Computing Environment", *ACM Operating Systems Review*, Vol. 21, No. 5, November 1987.
- [9] C. Pu, et. al., "The Synthesis System", *Technical Report No: CUCS-259-87*, Columbia University, 1987.

It is a pleasure to have you here today. The purpose of this meeting is to discuss the progress of the project and to plan for the future. We will be looking at the work that has been done so far and the challenges that lie ahead. I hope that you will find this meeting informative and useful.

The first item on the agenda is the report on the progress of the project. This report will be presented by the project manager, who will outline the work that has been completed to date and the results of the various tasks. This will be followed by a discussion of the challenges that the project is currently facing and the strategies that are being employed to address these challenges. The final item on the agenda is a general discussion of the project and the future. This will be an opportunity for everyone to share their thoughts and ideas on the project and to discuss any other issues that may arise.



A UNIX Implementation of HEMS

Craig Partridge
NSF Network Service Center (NNSC)
c/o BBN Laboratories Inc.
10 Moulton St
Cambridge MA 02238
craig@nnsf.net

ABSTRACT

The High-Level Entity Management System (HEMS) is currently the best known of the network management schemes designed to work on TCP-IP networks. In this paper, one of its designers describes experience with the first HEMS implementation, done under 4.3BSD.

Introduction

In late 1986, a number of researchers, users and vendors came to the conclusion that the TCP-IP protocol suite needed a standard internetwork management protocol and that none of the then existing management protocols was a suitable candidate for standardization¹. Working groups were formed to try to develop new management systems and protocols which could be considered for standardization. The High-Level Entity Management System (HEMS) is the best known of the systems to come out of this effort.

This is a description of the first HEMS implementation. The goals of this implementation were (1) to examine HEMS from a programmer's standpoint (instead of a system designer's); (2) to confirm that HEMS could be implemented; and (3) to acquire some performance information. The 4.3BSD system was chosen as the host environment for the implementation because it is a well-known operating system with good support for IP networking.

The presentation is broken up into three parts, corresponding with the goals of the implementation. The first section illustrates how one might use HEMS. (Readers interested in a detailed specification or a study of motivation are encouraged to read the references [1,2]). The second section describes the architecture of the implementation, and can probably be skipped by non-implementors. The third major section describes HEMS performance. Finally, in the conclusion, I try to summarize the current state of HEMS.

Overview of HEMS

HEMS is designed to provide the most critical network management function: a system-independent mechanism for performing monitoring and control of remote internetwork nodes. To put that another way,

HEMS makes it possible to observe and manipulate remote nodes on an IP network. Such a capability is essential if effective internetwork management tools are to be developed, although HEMS does not deal with how to build the tools themselves. One can think of HEMS as analogous to the *ptrace()* system call. One needs something like *ptrace* to be able to write debuggers such as *sdb*, *adb* and *dbx*. Similarly one needs something like HEMS to be able to write applications that do network management.

In HEMS, every IP node in a network (e.g., a host, terminal server or gateway) supports a hierarchical database which presents an abstract interface to the internals of the node. To perform management operations, applications send database queries to a query processor, or *agent*, on the node to be managed. The query language used in these queries is designed to be inexpensive to process. Query language operations which write into the database are mapped into operations which change the internal state of the node. So for example, changing the section of the database which represents the system routing table causes changes in the actual system routing table. Operations which read the database are mapped into operations which extract information from the node. Queries and the replies to queries are formatted in the self-describing data format, Abstract Syntax Notation 1 (ASN.1).

The meanings of query language operations and values in the database are standardized. The same database item retrieved from two nodes will have the same meaning and format and the values will be comparable, even if the nodes are made by different manufacturers. For example, the portion of the abstract database corresponding to the routing table looks the same on every node, regardless of how the actual routing table in the node is implemented. The goal is to create a system where a network manager trying to diagnose a problem can send queries to the node without having to know anything about the

¹Readers interested in network management issues are encouraged to join the network management mailing list, gwmon@sh.cs.net.

node other than its IP address.

HEMS can be used to manage all levels of the network stack in a node from the interface layer up to applications such as *telnet* and *rlogin*. Proxy and translator agents can be used to manage network components such as bridges, repeaters and modems, which may not have IP addresses. Proxy agents are agents which reside on a node with an IP address, which are designated as responsible for a component. A good example of a proxy is an IP node to which the control line of a modem is attached. The IP node is the only node which can effect control operations on the modem. Translators have a non-exclusive responsibility for a node. For example, a bridge might be managed through any one of several IP nodes, each of which is capable of controlling the bridge by translating HEMS requests into some link-level management protocol.

Details of HEMS: Four Examples

This section presents four examples, designed to illustrate how HEMS can be used.

A Small Database

For simplicity, these examples assume that we are attempting to remotely manage a network node that supports the simple abstract database shown in Figure #1. (The actual database proposed for use on IP nodes is much more complex and contains over two hundred separate items).

The database contains three items: a status register, the routing table (which contains the individual routing entries) and a routing distance indicator.

The status register is a virtual hardware status

register. In the examples we use it to allow us to reboot the node by writing the value 0 into it.

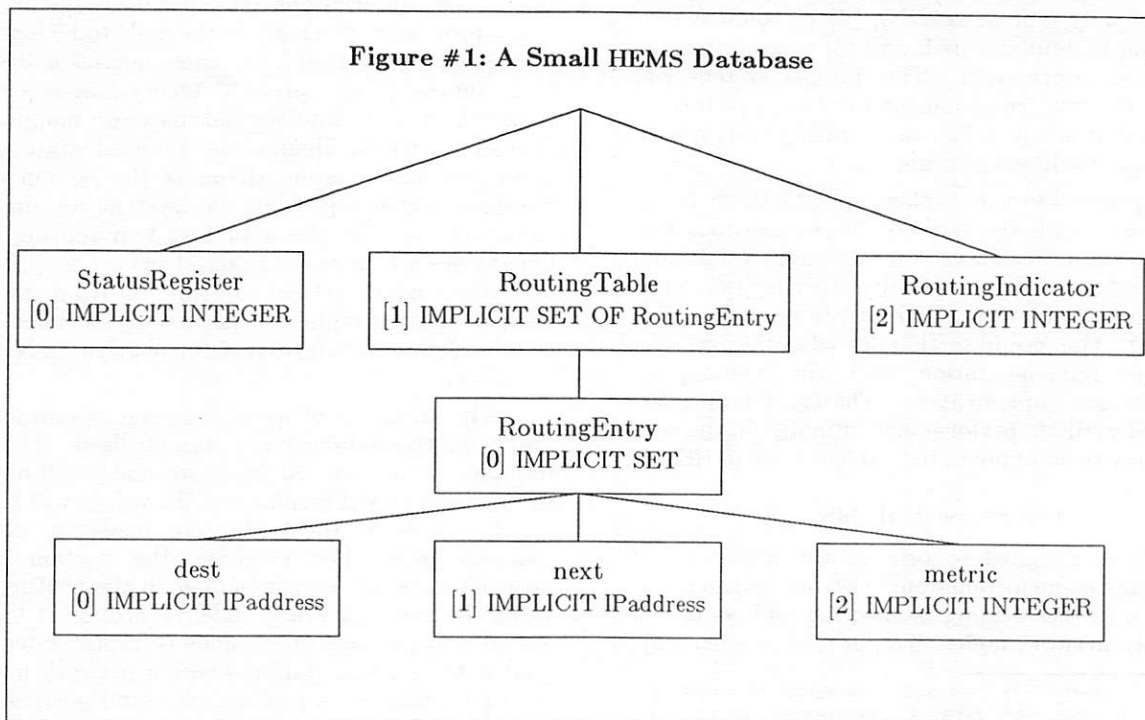
The routing table is a set of individual routing entries, and corresponds to the actual routing table used by the node. Each routing entry contains three items of information: the destination network or host (dest), the next hop gateway (next), and the distance to the destination using some metric (metric).

The routing distance indicator tells us how the node measures distance to a network. For the purposes of this example, we assume that two different distance measurement techniques are possible: a delay-based scheme (i.e., a network is so many milliseconds away), and a hop-count scheme (a packet must traverse a certain number of gateways to reach the network). If the delay-based scheme is in use, then the routing indicator has the value 1, and the distance metric in the routing entries is interpreted as a time value. If the hop-count scheme is in use, the routing indicator is set to 2, and the distance metric is the number of hops.

One important point about the database is its tree-shaped structure. The HEMS query language requires that any database it manipulates be a tree. This database schema was chosen because it is faster and easier to manipulate than a relational schema, and we were interested in reducing processing costs at the node.

Abstract Syntax Notation 1 (ASN.1)

Because HEMS requires that data from any node be intelligible to any other node, it needs to use an external data formats. External data formats are standardized data formats that all nodes use to



exchange information; if a node's data formats are not the same as the external format, it must convert its data into external form before sending it to any other node. HEMS uses the OSI external data format, Abstract Syntax Notation 1 (ASN.1) [3,4].

ASN.1 is a tagged data format. Each item of data is a triple of the form <tag,length,data>. The tag is a number that encodes information about the data (its type, whether it is a structure, etc.). The length is the length of the data section. There is no limit on the size of the length or tag fields. The fields are both self-describing and can contain arbitrarily large numbers. The data is either a value (e.g., the number 10 or the character 'a') or a collection of ASN.1 objects. Because ASN.1 objects can contain other ASN.1 objects, the data format can support extremely complex data structures including unordered collections of data, in which each item is identified by its unique tag.

An advantage of tagged data is that the sender and receiver do not have to agree in advance on the format of the data. Untagged data formats require the receiver to know the datatypes the sender will transmitting (for example, two integers followed by a character string). That was too restrictive for the

query language envisioned for HEMS. We wanted to use query language operations with variable numbers of operands, and needed tagged types to help identify each operand.

Figure #1 includes the ASN.1 type for all data items and some of the examples below show the binary ASN.1 encoding of the values. But for readability the examples are primarily written in a symbolic notation. In this notation, a name, such as "RoutingEntry", represents an ASN.1 data tag with no data; a name followed by a value in parenthesis is an ASN.1 object with a value (for example, "metric(10)", is a metric field with a value of 10); and a name followed by brackets such as "RoutingEntry{metric,next}", is a nested type, with individual fields listed within the brackets.

Example 1: Finding A Particular Route.

Assume that we are trying to send data through a gateway to a remote network (192.5.58.0), and for some reason, the data does not seem to be getting through. After confirming that our host is sending the packets to the gateway properly, we would like to check the routing tables in the gateway.

In HEMS, this is done by sending a HEMP query

Query #1: Getting The Routing Table	
Symbolic Notation	Hexadecimal Encoding
RoutingTable GET	8100410101

Query #2: Getting A Particular Route	
Symbolic Notation	Hexadecimal Encoding
RoutingTable BEGIN	8100410102
RoutingEntry Filter{item{equality{dest(192.5.58.0)}}} GET	80008006800380c0053a410101
END	410103

Reply #1: A Reply To Query #2	
Symbolic Notation	Hexadecimal Encoding
RoutingTable{	8112
RoutingEntry{	81000f
dest(192.5.58.0),next(128.89.0.92),metric(2)}	8003c0053a810480590005c820102

Reply #2: A Reply To Query #2	
Symbolic Notation	Hexadecimal Encoding
RoutingTable{	8124
RoutingEntry{	81000f
dest(192.5.58.0),next(128.89.0.92),metric(2)},	8003c0053a810480590005c820102
RoutingEntry{	81000f
dest(192.5.58.0),next(10.4.0.5),metric(1)}	8003c0053a81040a040005820101

message to the gateway. HEMP is the HEMS message protocol. A *query* message, is a particular type of HEMP message, which asks for information. All HEMP messages come in two parts, a standard header, which contains some basic information about the message (whether the body of the message is encrypted, what access permissions it has, an ID number, the message type, etc.), and the body of the message, which is a sequence of ASN.1 objects. The interpretation of the body is per-message-type specific.

For queries, the body of the message is an ASN.1-encoded database query, written in the HEMS query language. The language is a sequence of operands punctuated by operations on those operands (i.e., operations follow their operands). An operation may have a variable number of operands. The operations can retrieve data, change data, delete or add data, or manipulate the context stack that is used to keep track of what part of the database is being accessed.

Looking at our example, what we want to do is retrieve a route, so we would use the GET operation to extract the route from the database. There are several ways to do this. The simplest request is shown in Query #1, which extracts all the RoutingEntries which make up the routing table and returns them to our application. We then would have to read the entire table to find the particular route of interest. This is a rather expensive approach; if the node is of any importance, its routing table is likely to contain a few hundred entries. The cost of dumping all those entries onto the network and then processing them at our end to find a single route is clearly excessive.

To allow a remote user to retrieve selected items of information, HEMS supports *filters*, a method for selecting items using boolean expressions. In our example, we wanted the RoutingEntry which has a destination network of 192.5.58.0, so we can use a filter on the *dest* field, as shown in Query #2.

In this case, the GET command takes two arguments, the data to retrieve (a complex RoutingEntry) and the filter that tells us which RoutingEntry is the one we want. Note that we've also introduced two other operations, the BEGIN and END operations. Filters only work on sub-nodes of the current node of the tree, so we have to move ourselves into the node from which we want to apply the filter. BEGIN is the HEMS version of *pushd*, and moves us to a new context by manipulating the context stack; END moves us back to our previous context by popping the context stack. Query #2 is probably the one an application would use.

When the node processes the HEMS query, it sends back a HEMP *reply* message. The body of the reply message is the answer to the query. In the case of Query #2, if the route existed and went through gateway 128.89.0.92, the reply might be the one shown in Reply #1. Reply #2 is an example of a reply where two routes existed. If no route existed, the reply would be an empty RoutingTable structure.

Note a nice feature of these replies. They represent walks of the database tree. In fact, all HEMS replies are a walk of the database tree (possibly with multiple visits to the same node).

Also keep in mind that the form of the replies is the same regardless of what type of node is queried. The node is required to convert from its internal format, such as the 4.3BSD *rtentry* struct, into the ASN.1 structure used by HEMS.

Example 2: Rebooting A Node

Example #1 showed how one can do monitoring with HEMS. In this example we look at the more complex problem of control (being able to change how a node behaves).

Query #3: Rebooting a Node	
Symbolic Notation	Hexadecimal Encoding
StatusRegister(0) SET	800100410108

Imagine that we want to reboot a remote node. The HEMS query to tell a node to reboot itself is simple. We need only load the value 0 into the status register, using the SET operation, as in Query #3. When a node receives this query, it is expected to take whatever action is required to cause itself to reboot. For example, on a BSD host, the HEMS agent would invoke the *reboot()* system call. On a machine where the agent ran in privileged mode, it might just write a bit in a processor status word.

The complexity of control operations is not in the operations themselves; the SET, ADD and DELETE operations are like all other operations. The problem is what the operations allow a remote user to do. It is one thing to allow a remote user to retrieve monitoring data; it is quite another to allow remote user to change how a box behaves. Support for control operations all but requires support of access control and encryption mechanisms.

In HEMS, these facilities are provided by HEMP. At the start of each HEMP message are optional sections for authentication and encryption information. The authentication section allows remote users to prove their membership in one or more access groups. We preferred per-message authentication to per-operation authentication because we felt it was more efficient to process authentication information once at the start of a message. The encryption sections allow remote users to indicate what encryption method they are using in their message and what type of encryption they would like in the reply.

Example 3: Learning That A Node Is About to Reboot

There is something left out of Example #2. What happens if we are not the only people watching the node we reboot? How do we notify everyone that this reboot is planned and not the sign of hardware or software problems?

It turns out that the best way to notify everyone

is to have the node tell them. When a node receives instructions to reboot, it sends out a HEMP *event* message, which notifies interested managers that the node is about to reboot. Event messages look like unsolicited reply messages. They contain an indication of the type of event which has occurred and any relevant data from the agent's database. Managers that care about whether a reboot is planned can deposit their address with the HEMS agent. Such requests are made on a per-event basis; a manager can select the events it wants to receive.

Like everything else in HEMS, events are standardized. The same reboot message is sent from every node when it is about to reboot. One of the outstanding pieces of work left for HEMS is defining all the possible events. Research indicates that there are probably about 500 to 1,000 distinct events that need to be defined, a massive task.

Example 4: Changing The Routing Configuration.

Finally, we look at a more complex HEMS scenario. Imagine that we have misconfigured a node to use delay-based routing metrics when all other nodes in our network use hop-count metrics. As a result, our node is distributing misleading routing information to the other nodes and is misinterpreting the routing information it receives. We would like to fix the routing metric and also look at the node's routing table to see how badly corrupted it is. A query to do this is shown in Query #4.

Query #4: Fixing the Distance Metric	
Symbolic Notation	Hexadecimal Encoding
RoutingTable GET	8100410101
RoutingIndicator(2) SET	820102410108

This query extracts a copy of the routing table and then sets the routing indicator to use hop-count metrics. Because the routing indicator is a key variable, the node would probably send an event message when it is changed. Notice that HEMS allows us to mix monitoring operations with control operations in a single message. This is one of the unique features of HEMS. The other major management protocols do not support this feature [2].

The Implementation

This section is a detailed look at the implementation of the HEMS agent under 4.3BSD. The purpose is to illustrate one way an agent might be implemented. A question sometimes raised by those who read the HEMS specifications is whether HEMS is too complex to reliably implement. I believe this implementation shows that the HEMS is not extraordinary in its complexity.

To limit the complexity of the system, the implementation breaks the agent down into three distinct modules: an ASN.1 preprocessor, that converts inbound and outbound streams of ASN.1 data into a convenient internal form; a HEMP message processor, that reads and generates HEMP messages; a query processor which reads and interprets the query language sent in queries using a database structure which allows the query processor to map between ASN.1 objects and kernel data structures. We look at each module in detail below.

HEMP Message Processor

Whenever a HEMP query is received over the network, the new query is passed to the message processor. The message processor reads the header of the message, which involves checking version numbers, processing any authentication mechanisms, and, if necessary, enabling encryption and decryption. Finally, it generates the header of the reply message and calls the query processor to handle the body of the query.

Query Processor and Abstract Database

The query processor does most of the work in the HEMS agent. At its heart is the short loop shown in Figure #2. There is only one hard step in the loop: doing the work to perform the operation.

To perform an operation the agent needs to solve two problems. First it needs to properly interpret the operands. Second, it must effect the operation on the database and the host node.

Interpreting the operands was the harder problem. Every operand must be interpreted in the current context in the database tree because the ASN.1 tags which identify the values are context-specific. Furthermore, the operands can be complex structures. For example, Query #5 instructs the agent to change two of the fields in the selected RoutingEntry and leave the other fields untouched. There is a tricky data structure problem here. We need to be able to

Query #5: A Complex Operand	
Symbolic Notation	Hexadecimal Encoding
RoutingTable BEGIN	8100410102
RoutingEntry{metric(1),next(128.89.0.1)}	8009820101810480590001
Filter{item{equality{dest(192.5.58.0)}}} SET	8006800380c0053a410108
END	410103

combine information from the context stack, the database and the operands to determine where the operation is to be applied.

Figure #2: Query Processor

```
WHILE there are more ASN.1 objects
  READ the next ASN.1 object
  IF the object is an operation
    do the operation
  ELSE the object is an operand and
    save it for the next operation.
```

The implementation solves this problem by using the same data structure for the database, the operands and the context stack. All three are represented by a tree structure, which mimics the nesting of the ASN.1 objects. So the RoutingEntry operand of the last example, would be represented as a tree of depth two, with the root representing the RoutingEntry, and having two children, one for the metric field and one for the next hop field. The context stack is a pointer into the tree structure which implements the HEMS database. The implementation then uses a set of routines which can use one structure as the template for applying an operation to another structure. Again, looking at Query #5, once the filter has located the proper RoutingEntry to change, the agent calls a routine with the operand containing the fields to be changed, and the RoutingEntry we wanted to change. The routine makes sure that the SET operation is applied to the metric and next hop fields of the old RoutingEntry. Note that support of filters is also made easier by this method. The tree structure of the data makes a good parse tree to hand to an expression handler; filters do not have to be further processed before they are evaluated.

This brings up the second problem: what does it mean to do a SET on an object? (Or a GET, ADD, or DELETE for that matter). Because the database is an abstraction of the host node, reading from it or changing it is not a matter of simply changing the database. The agent must change the state of the node to correspond to changes in the database.

There are at least three approaches: (1) One can maintain a full database (possibly on disk) and run a background daemon that keeps the database and the actual host software consistent. The major problems with this approach are that if the daemon fails, the remote manager may not see this because the database will still be accessible, and that running such a daemon is expensive given that one expects only a subset of the database to be accessed on a regular basis. (2) One can maintain a full database but integrate it tightly into the host software. So, for example, the code which maintains the routing section of the database is also the same code that accepts

routing updates from protocols and chooses routes for datagrams. This approach makes a lot of sense but requires a major rewriting of the host software. Or, (3) one can maintain a database of entry points into the host software. Except for a few static values, the database contains no real data, just methods for manipulating the host software based on remote requests. So for example, on the BSD system, when a remote request to read the routing table is received, the database reads */dev/kmem*, and repackages the BSD routing structures into ASN.1. A request to change the routing table is translated into a sequence of *ioctl()* calls. This is the approach used in the implementation. It works reasonably well, except for the nuisance of having to write special access routines for portions of the database.

ASN.1 Preprocessor

One the hardest problems in the implementation was finding a way to limit the amount of input processing that needed to be done. The agent reads from a stream of ASN.1 objects. This stream may be encrypted, and it may be necessary to buffer the data. Furthermore, ASN.1 uses variable length fields, so processing the objects in their ASN.1 format more than once is undesirable.

One option is to try to fully process each object as it is received. This approach doesn't work well in HEMS. Query language operands really need to be processed at least twice; once upon receipt, when they are scanned and stored waiting for the operation, and again when they are actually used by the operation. Filters are accessed repeatedly during processing. So the implementation uses another approach, that of converting each object as it is received into the internal format described above. The ASN.1 preprocessor is responsible for doing the conversions to and from ASN.1. It also quietly does encryption and decryption of data streams.

Modules which are reading data, call on the routine *asn_read()*, which returns the tree-shaped structure which is the internal representation of the next ASN.1 object in the input stream. This internal representation incorporates all ASN.1 tags and length information, but maintains it in an easy-to-access format.

Modules which are writing data call on the routine *asn_write()* which takes the tree-shaped internal representation and generates and sends the ASN.1 representation.

Both *asn_read()* and *asn_write()* quietly do data encryption and I/O buffering. Another routine, *asn_flush()* is invoked at the end of all processing to force transmission of any partially buffered or encrypted data.

Performance

One of the key goals of HEMS was to design a system that did not overburden the machines it ran on. The view we took when designing HEMS was that it should be possible to develop a full function inter-network management protocol, without consuming huge amounts of memory space or CPU processing. We wanted it to be possible to run HEMS on the 68000-based IP routers that have recently become popular.

To gather information on whether HEMS met these goals, several tests were performed with this implementation on a Sun-3/50 workstation. These tests were not intended to be definitive, because the implementation is not highly optimized. The goal was to gather some information which would tell us whether HEMS was likely to meet its goals. This section discusses the results of the tests and what they tell us about HEMS' performance.

Code Size

Currently the HEMS agent has a running image size of about 83Kbytes (this after sending a few queries to it). That is a little bit bigger than we had hoped. When we were originally designing HEMS, an unofficial goal was an image size of no more than about 100Kbytes. Since the implementation is incomplete (in particular, event handling and some parts of the database are not yet supported) it is not clear that this particular implementation will meet that goal. On the other hand, since the implementation is nearly complete, and could presumably be optimized to use less data, we probably won't miss our target size by too much. Other implementation approaches may also yield more compact binaries.

Processing Costs

Two sets of tests were run to measure the cost of processing HEMP query messages. Both tests used the *gprof* profiler to get performance times.

The tests tried to estimate the cost of receiving and processing a query. Two sample queries were developed. A short query which retrieved two values (a BEGIN, two GETs and an END instruction) and a medium-length query which extracted the same information plus the entire routing table (which contained

10 routes). Each query was sent to the agent one thousand times per profiling run. The profiles were then analyzed to determine cost of processing the queries, and also to estimate which modules of the implementation were more expensive.

The per-query costs are shown in Table #1. The costs have been broken down to show processing costs for HEMP and the query language separately. Note that these numbers are not terribly precise. *Gprof* showed considerable variation (about 15-25%) in processing times from one test to another. As one indication, the HEMP header was the same for both queries, so the HEMP processing time should have been the same. Instead they differ by 25%.

Viewing the numbers as an informed guess, they still look pretty good. Current experience with Internet gateways suggests that the average gateway receives a query once every few minutes, and in peak traffic periods might get as many as two or three a second. Using the numbers for the medium size query, this suggests that at a peak load of three medium sized packets per second, only 15% of the gateway's processing time would be consumed with management requests.²

The *gprof* output was also screened to try to determine where the agent spent most of its processing time, by module of the software. This information is shown in Table #2. (The miscellaneous category represents initialization costs, network connection set up, etc.).

The table suggests that the majority of the processing time is spent in the ASN.1 preprocessor, reading and writing ASN.1 objects. Further examination of the the profiling output suggests the majority of costs are spent in UNIX system calls (e.g., *read* and *write*) not in the ASN.1 parser. Some of this expensive is may be due to the fact that the agent does single character reads for the first few characters in each HEMP message, until it has confirmed that the message is not encrypted.

²Note that this 15% figure should not be taken terribly seriously. Your *gprof* may vary. Furthermore, gateways may be more sensitive to context switching and query size than a BSD application.

Table #1: Average Query Processing Costs

(in milliseconds)

Query	HEMP Processing	Query Language Processing	Total Processing
Small	8.1	19.8	27.9
Medium	6.1	41.2	47.3

Table #2: Approximate Processing Time By Code Module

(in percent)

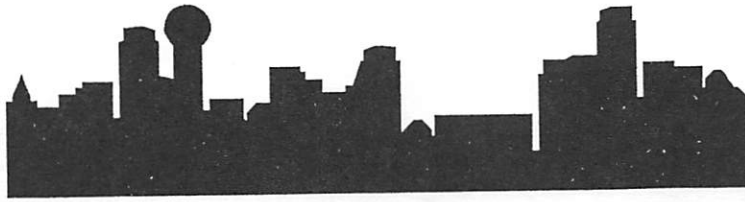
ASN.1 Preprocessor	Query Processor	HEMP Processor	Misc
62%-75%	6%-32%	1%-2%	5%-17%

Conclusions

The goals of this implementation was to show that HEMS can be implemented, and that efficient remote management of network nodes is possible. I believe that those goals have been achieved. HEMS can be implemented, and the limited testing so far suggests that it is not very expensive to process.

Bibliography

- [1] C. Partridge and G. Trewitt, The High-Level Entity Management System (HEMS); RFCs 1021-1024. In *Network Working Group Request for Comments, no. 1021-1024*, Network Information Group (NIC), SRI International, Menlo Park, Calif., Oct. 1987.
- [2] *IEEE Network*, Vol 2, no. 2. Special Issue on Internetwork Management Protocols. March 1988.
- [3] *Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)*. International Standard, no. 8824. International Organization for Standards, May 1987.
- [4] *Information processing systems - Open Systems Interconnection - Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*. International Standard, no. 8825. International Organization for Standards, May 1987.



Building an Equities Trading System in a Distributed UNIX Environment

Mark Luppi (mwl@mstan.uu.net)
Mark Seiden (mis@mstan.uu.net)
Joseph Collins
Daniel Fisher
Keith Iverson
Charles Marshall
Josef Sachs
David Shaw
Morgan Stanley & Co.
1251 Avenue of the Americas
New York, New York 10020

ABSTRACT

This paper discusses our experience in building a distributed equities trading application in a UNIX local area network. The key problem (and the primary subject of the paper) is ensuring that trading operations are never interrupted for more than a few minutes: extended downtime can leave existing stock positions badly exposed.

Providing high availability for single-machine applications is a relatively solvable problem (there are a variety of fault-tolerant systems to choose from). Protecting against extended outages of distributed LAN-based applications is much more difficult. Although there is a wealth of literature on this subject, we found no currently available UNIX products that offer a comprehensive solution to this problem.

We considered and rejected the idea of building some form of dynamic distributed recovery management ourselves, adopting instead a simpler approach: the use of fully replicated networks, one a "primary" network where the traders are active, the other a "standby" network. If the primary network fails, the traders transfer to the standby side. This is an old idea, but with a hard problem at its core: keeping the state of the two networks synchronized, so that operations can resume quickly upon the standby network.

This paper describes the duplicated-network architecture, particularly our method for keeping the two networks synchronized by periodic transfers of state information, and discusses advantages and disadvantages of alternative synchronization methods. This should be viewed as an "experience" paper, which we hope will be useful to those concerned with availability and recoverability of their distributed applications.

Equities trading and UNIX systems

Equities trading can be viewed as a special breed of on-line transaction processing (OLTP). It is highly interactive in nature: traders monitor their current stock positions and issue buy-sell transactions relative to markets that are in constant flux. Over the past thirty years the process has become increasingly computerized, and this has profoundly changed the psychology of the business. The average turnaround time of a transaction, from initiation (when new market data arrives) through decision (when a trader's buy or sell order is placed) to completion (when confirmations are received that the order has been filled) has in many environments shrunk to seconds. Buy-sell orders are increasingly placed over automated links to the exchanges; information about these transactions is available moments after they

occur. At present, human intervention is still required to fill the orders once they reach the floors of many exchanges, but the Toronto Stock Exchange's recent introduction of a system that automatically matches buyers and sellers is indicative of the future.

This shrinkage in reaction time has been accompanied by a rapid growth in the sophistication of trading strategies. Increasingly, traders are using statistical, numerical, heuristic or artificial intelligence techniques for analysis of complexly related market factors. These analytics must often be performed in real time, filtering large volumes of market data to identify trigger conditions for stock transactions. "Program trading" techniques (recently placed in the limelight) are only one form of this phenomenon.

The rapid evolution of trading systems accounts for much of Wall Street's recent interest in UNIX

systems. Historically, trading-system designers have been skeptical about the reliability and general suitability of UNIX for financial OLTP applications. Some new developments have begun to change this attitude:

- i. *Availability of high-performance systems.* UNIX has achieved widespread acceptance among vendors of supercomputers and "mini-supers." Wall Street firms are increasingly interested in these machines since they offer attractive price/performance and great flexibility for designing compute-server architectures in a local area network environment.
- ii. *Portability/Standards.* Rapid changes in trading technology have increased the importance of portability and standardization. In our opinion, a UNIX-based LAN, tying together workstations, data servers, compute and communication servers, offers the best chance of incorporating new and appropriate technology without undergoing a massive conversion and integration effort at each evolutionary step.
- iii. *Recent DBMS Products:* The advent of reasonably performing database management systems makes UNIX a more attractive choice for OLTP applications. A commercial DBMS can provide the OLTP mechanisms found in Tandem and IBM's CICS but missing from UNIX systems. These mechanisms include transaction journaling and commit/abort processing with automatic rollback of aborted transactions. Rather than building this kind of transaction management into the application, the OLTP developer has the more palatable option of letting the DBMS do it. In addition, most of the DBMS's bypass the UNIX file system, which provides a stronger guarantee of real time performance.

Reliability, however, remains a major concern. This is particularly true for our trading applications, which essentially cannot have extended outages. We can tolerate occasional system failures, but we must *always* resume operations within a few minutes.

"Always" is a hard word in this context. Some system failures¹ can be cleared quickly by rebooting or by replacing a board; other failures may take hours to repair. Preventing extended downtime is a relatively achievable goal for single-machine applications. (One can use fault-tolerant machines which "guarantee" a program's continuance). It is much harder to achieve for distributed applications on a local area network of heterogeneous machines. Although there has been extensive research on this subject [1], there are few if any solutions available in the form of vendor products.

In the end, we had to piece together our own solution, using parts from vendors. Our intent was to build a system which would be remain highly

available despite failures of individual components (whether hardware or software). We now survey the advantages and drawbacks of a number of approaches that we considered, and describe the scheme that we ultimately implemented. Our primary focus has been to answer the question: "How does one achieve a high level of availability for a distributed OLTP application using existing UNIX products?" We use the term "high availability" throughout the paper to characterize systems where occasional failures can occur, but where the system must recover quickly enough so that operations are not interrupted for more than a few minutes. (This is a less stringent requirement than true "fault tolerance," but still difficult to achieve).

System implementation

System description

A team was formed at Morgan Stanley & Co. to design our systems based on a BSD UNIX network of machines using TCP/IP, which we believe provides the best foundation in terms of richness of function and maturity. Our starting point was an OLTP trading system that had been written in APL for an IBM mainframe. We intended to use this software as a test case to build a generic distributed framework that can support a broad range of future applications.

We found that the typical functionality of a trading system maps cleanly into a distributed environment. What existed as separate modules on the mainframe became server programs which would be placed on the various nodes on our network. The roles played by individual nodes include:

- i. *Trader workstations:* presentation services (menus and graphical displays), initiation of transactions (queries and buy/sell entries).
- ii. *Compute servers:* filtering and analysis of market data in real time.
- iii. *Communication nodes:* external interfaces to our network, including real-time market datafeeds, and order-execution channels for trader buy/sell requests.
- iv. *Database and file servers.*

The initial machines on our network are Sun-3 and Sun-4 servers, Sun-3 diskless workstations and an Alliant FX/8 mini-supercomputer. We anticipate that machines from a number of other vendors will be added. Likely candidates include fault-tolerant data servers and additional compute servers.

Achieving high availability

We knew that our hardest problem would be to ensure a high level of availability for our distributed application. Our requirements were stringent: occasional outages could be tolerated, but trading operations must always resume fully within fifteen minutes. (The target is actually five minutes, with fifteen minutes as a "worst-case" upper limit). Much of this time is required by the traders to run a gamut of validation checks after recovery. This leaves little time

¹We use the term "failure" to refer to functional impairment caused by a hardware or software problem, and won't discuss here provisions for handling environmental failures (power outages, air-conditioning failures, etc.).

for fault isolation and repair, nor could we afford a lengthy re-initialization to bring the system back into operation.

One way of increasing network reliability is the use of highly reliable components (e.g., MilSpec parts and fault-tolerant machines). This approach, however, could only take us so far. The workstations and compute servers that we wanted to use are not built with high reliability in mind. (We're resigned to the likelihood that higher performance, higher function, state-of-the-art hardware usually will have a higher level of flakiness). We therefore needed a global strategy to protect the entire network.

It would be nice, of course, if the network could manage its own recovery dynamically, with automatic detection of failures and rapid reconfiguration around the failed component. In practice, no vendor of UNIX products provides a cost-effective generic solution of this sort for commercial distributed applications.

We considered and rejected the possibility of building some form of dynamic distributed recovery management ourselves. We were skeptical that a comprehensive solution was even feasible; in any case, it would (i) be difficult to implement, (ii) cost a lot, and (iii) add substantial complexity to our environment. In the end it would be hard to prove that we had adequately defended against all likely (and unlikely) failure scenarios.

Duplicated networks

We decided that using replicated networks was the best means of achieving our main goal of short recovery time. At any given time, one network runs the trading application and is designated the "hot side" or primary network; a second network serves as a "warm" standby. If the primary network fails, trading activity moves to the standby network.

All critical components are duplicated on both networks, including analytic engines, communication servers, database servers, and workstations. When the primary network fails, each trader transfers by switching his or her display monitor to an alternate workstation on the standby network. We could contemplate such massive replication since minimizing cost was not our foremost concern.

Using duplicated networks to achieve high availability is an old idea. It has one outstanding advantage: there is no need to protect against all conceivable error conditions. If a failure occurs that cannot be quickly diagnosed and repaired, one can "abandon" the primary network and move to the standby.

The difficulty with this approach is ensuring a *rapid* transfer to the standby side. We could not afford a lengthy initialization to bring the standby network into operation. At the same time we had to provide full functionality when the switch occurred, with immediate access to the entire record of that day's trading activity. This ruled out traditional but time-consuming recovery mechanisms such as playing back the system's transaction logs from a start-of-day

checkpoint. (Since our application may eventually handle thousands of transactions each day, such a playback would simply take too long).

Synchronization of the two networks

It was clear that we would have to keep the state of the two networks continuously synchronized. Attempting this synchronization in real time is a messy proposition: there are many windows and edge conditions that can trip the unwary. We were willing to constrain our application architecture to simplify the synchronization problem:

- i. Applications maintain their "critical state" in a single database, including all data and status necessary for an immediate resumption of operations.
- ii. It is acceptable to lose that few seconds of transactions occurring immediately before a crash. These transactions can be restored manually when operations resume on the standby side. (This restoration adds another step to the traders' validation procedures, but should not unduly lengthen the recovery process).

Even with these constraints, we found no "off-the-shelf" vendor products that could meet our requirements for this synchronization. There were products, however, that provided us a base for "rolling our own" solution. We considered a number of methods which fell into two general categories:

- i. *Sharing a single image of the database:* Two or more networks attached to a dual-ported disk or to members of a cluster (e.g., VAX or Tandem) can access a common image of a database containing all necessary operational information. If the primary network fails, the standby cleans up the situation and continues based on the contents of the database image.
- ii. *Shadowing the primary database:* In this scheme, system state resides in a central database on the primary network; changes to the database are continuously transferred through to a database on the standby network. This shadowing can be accomplished by (1) requiring the application to send each update simultaneously to both databases; or (2) periodically transferring the state of the primary database through a controlled gateway or other communications link to the standby side.

For reasons described in the following sections, we chose to periodically transfer the state of the primary database to the standby.

Periodic transfer

In this scheme, changes to the primary database are applied periodically to the standby database. A daemon process captures the most recent updates to the primary database and transfers them to a cooperating process on the standby network. The second process then loads these updates into the standby database.

We preferred this approach for a number of reasons.

- i. *Introduces the least interaction and hardware connectivity between the two network sides (a controlled network gateway) and avoids any single point of hardware failure.* The gateway itself is not likely to be a single point of failure for the whole network, since even if it goes down, the primary network can still function.
- ii. *Robust against software bugs causing data corruption.* Since the "shadow-copy daemon" only extracts and sends "well-formed" transactions, corruption of the primary database's internal structures by DBMS or even by many kernel bugs will not affect the standby.
- iii. *Ensures that the two databases stay largely in synch.* If the primary fails, only those transactions committed since the last periodic transfer are unavailable on the standby.
- iv. *Relatively easy to implement.* This is true if the DBMS provides the ability to capture a "snapshot" of all committed transactions in the database at a particular point in time. Otherwise this method is extremely hard to implement in a way that preserves the consistency of the standby database (the reasons are discussed below).

There is one obvious disadvantage. If the primary fails, those transactions occurring since the last periodic transfer will not be available on the standby side, and will have to be manually restored. The extent of this loss depends on the frequency of the transfer.

Implementation issues for periodic transfer

To implement the "periodic-transfer" scheme, it is necessary to capture a "snapshot" of the database which includes all committed transactions until then. If a piecemeal transfer were done (i.e., periodically copying the database a table at a time), the standby database could be inconsistent, as the following scenario demonstrates:

- a. The daemon copies table A to the standby database.
- b. A process on the primary network updates tables A and B as part of a single transaction.
- c. The daemon copies table B to the standby database.

Clearly, tables A and B in the standby database reflect two different states of the primary database. This inconsistency could cause trouble if the primary network were to crash immediately after step c. Using a single global lock for the entire primary database while copying it guarantees consistency but has a number of other problems:

- i. Accessing processes must be well-behaved and acquire the lock before each update transaction; this will cause some performance degradation.

Some form of lock management is also necessary to exclude the shadow-copy daemon while allowing all other processes to share the lock.

- ii. Since pending transactions could hold the lock for an indeterminate time (maybe hours), it would be difficult to guarantee the periodicity of the copy.
- iii. There will be a noticeable system slowdown when all accessing processes wait while the entire database is copied.

The problems with a table-by-table (or block-by-block) copy should be clear. It is essential that the DBMS be responsible for the critical step of "snapshotting" the database, using some low-overhead method that preserves global consistency. The daemon can then move the snapshot image, to be applied to the standby database.

Somewhat to our surprise, we found that most DBMS vendors do not currently provide the ability to "snapshot" a consistent database image. We selected Sybase Inc.'s RDBMS product in large part because it does have this capability. We liked a number of other things about Sybase: it provides strong performance, along with useful features like stored SQL procedures and triggers for referential integrity [2].

The Sybase solution

Sybase provides four operations for database "snapshots." These operations can be included in C programs as part of SQL transactions [2].

- i. *DUMP DATABASE.* This dumps an entire database, including all tables and the transaction log to an ordinary file.
- ii. *LOAD DATABASE.* This creates a database image from the dump file produced by an earlier DUMP DATABASE. If the dump contains uncommitted transactions, the LOAD operation rolls them back.
- iii. *DUMP TRANSACTION LOG.* This operation has substantially less overhead than a DUMP DATABASE. It dumps to an ordinary file all committed transactions recorded within the database's transaction log. These transaction records are then removed from the log, so that the next DUMP TRANSACTION LOG will not contain them.
- iv. *LOAD TRANSACTION LOG.* This operation updates a database image by applying the output of an earlier DUMP TRANSACTION LOG.

If a transaction which includes a DUMP operation aborts, the effects of the dump can be undone as part of normal rollback transaction processing. (This makes some of the error handling much easier). Sybase currently does not provide rollback for the LOAD operations.

The daemon that we have added synchronizes state between the primary and standby DBMS by shadow-copying the primary's dumped transaction log periodically to the standby network, where a

second process applies it to the standby database. It uses the low-overhead DUMP and LOAD TRANSACTION LOG operations. The more expensive DUMP and LOAD DATABASE operations are only used for the failure recovery situation.

Sun's implementation of RPC (remote procedure call) made this relatively easy. The primary daemon calls the standby and receives results using the RPC mechanism.

In extremely pseudo-code the daemon looks like Figure 1 (unrolled a bit for clarity).

The dump files currently reside (just as an experiment) on a NFS file system shared (writeable) by the primary and standby data servers. Use of a shared file system is by no means necessary – in this case, instead of copying we could *rcp* the dump files from an agreed-upon place on the primary to the standby. (Certainly the shared file system is an extra “entangling alliance” between the two sides that increases their interdependence. Could it cause one side to “infect” the other? Are there arguments to put any more than the minimum of state information in any sort of UNIX file system?)

An inelegance here is that we use both a belt (transaction logs) and suspenders (sending the whole database). Sybase restricts what we can do on the standby side: we can only load transaction logs in their proper time order, and when a LOAD TRANSACTION operation succeeds we can't roll it back. This means if the *notice* of success cannot be delivered (say due to transient network problems or warm side

failure), the primary side will see a timeout, and will roll back the DUMP TRANSACTION LOG; then the next transaction log will have a timestamp which the warm side will recognize to be “too early” and thus unloadable, forcing us to recover by transmitting the entire database. Luckily, transaction activity can continue on the primary side even during a database dump, and our databases are relatively small.

Allowing rollbacks of LOAD TRANSACTION would make it easier to extend this code to the multiple-network case (assuming we wanted to keep more than one standby network). There seems no easy way, however, to extend it for multiple primary-side databases (i.e., periodically copying two or more databases from the primary) while still preserving consistency on the standby network. Since the copy daemon can do a DUMP operation only on one database at a time, transactions can slip in during the window between successive instances of DUMP. Thus, transactions affecting multiple primary databases might be found in some of the standby copies but not in others.

Reaction to failures

What happens in the event of a primary-side failure? We attempt to determine whether it's a transient software failure (and restart the failing service) or a failure of a critical part that must force a switch to the backup. In cases where we cannot quickly detect and repair the failure our policy is to always switch.

A switch causes the system to be reconfigured. We first close the gateway between the primary and

Primary side:

```

register remotely-callable procedures;
forever
wait for an rpc;

forever
begin transaction;
dump transaction log to shared file;
rpc warm side("load transaction log");
    copy transaction log to a safe local place;
    load transaction log of standby db from safe copy;
    if success return("success");
    else return("send whole database");

if rpc failure or timeout
    rollback transaction;
else if rpc returned "success"
    commit transaction;
else if rpc returned "send whole database" {
    dump database to file;
    rpc warm side("load whole database"); }
    copy database dump to a safe local place;
    load whole database from safe copy of dump;
    if success return("success");
    else complain loudly and die;

wait until it's time to do it again;
```

Figure 1

standby networks. We then cool down the corpse of the hot side, sending deactivation messages to kill its servers. In parallel with this, we heat up the warm standby, sending activation messages to the replacement servers there (they are already initialized but quiescent). Meanwhile, traders can switch their displays to the workstation on the side which is now hot. There they perform a series of validation checks and any other user-level recovery actions, and continue with operations. After repair of the fault, the gateway between the two systems will be restored, and the corpse will be revived as the warm standby.

Transactions could be lost owing to a crash, but the exposure is limited to those which have occurred since the last shadow copy, which will happen every few seconds. Traders restore these transactions from a hardcopy log maintained during normal operations. (This log is already kept for the mainframe version of the application, and is used mainly to reconcile orders that have been placed but not yet reported filled). We're not happy about losing these transactions, but even if we captured every single transaction committed to the database, traders would still have to perform a similar check for transactions that were pending when the crash occurred.

Setting the interval for shadow-copy is a performance versus reliability tradeoff. We tuned this interval by running benchmarks of our application concurrently with the shadow copy daemon. Increasing the frequency of the DUMP TRANSACTION LOG decreases the average size of the transaction log to be copied, which results in less work per shadow-copy. We were pleasantly surprised to find that we could shadow-copy as frequently as every fifteen seconds without noticeably affecting performance.

Alternative approaches

In the following sections, we discuss some additional approaches to the synchronization problem.

Dual updates

We talked to several DBMS vendors about another method of maintaining duplicated databases: simultaneous updating of the databases on each network. In this approach, each time an application updates the primary database, it is responsible for applying the same update to the standby database.

While it would have been possible to modify our application to explicitly perform such paired updates, the preferred solution was to have the vendors include a multiple-update option in their application library routines. (These routines allow access to SQL facilities from C). Once the multiple-update option was enabled, it would be transparent to an ordinary application. We especially wanted error detection and recovery to be hidden by library routines. If a transaction aborted on one side, it should be rolled back on the other side (using two-phase commit or a similar mechanism). The user could define policy for retries of failed transactions, including:

- i. Number of retries for particular error conditions.
- ii. What to do if one side failed permanently (discontinue updates to the healthy side, or else continue updating that side only).

A possible mechanism for this (suggested by Sybase) is a user-supplied "retry-policy" routine called into action by SQL library routines when a failure occurs.

At first this approach seemed straightforward: to keep multiple copies, it should be simplest for the application to write simultaneously to multiple places. Closer investigation revealed that multiple updates are susceptible to some well-known complications of distributed DBMS processing. The following examples illustrate only a couple of the problems:

- i. *Partitioned network*: this is a generic problem for distributed DBMS [3]. In the context of multiple updates, it arises when some of the workstations think at least one of the databases has failed or otherwise gone away, while other workstations think the database is still there and continue updating it. Inconsistencies among the databases immediately arise.
- ii. *Data Sequencing*: transactions can be applied in differing order to each database. (For example, transaction A from one process could overwrite transaction B from another process in the primary database. This sequence could be reversed for the standby database, with transaction B overwriting transaction A).

There are new possibilities for deadlock as well. To solve these problems would require complicated mechanisms that the vendors do not currently provide. We therefore discarded this solution.

Sharing of dual-ported disk

Dual-ported disks provide another way to synchronize the networks. A single image of the database resides on a disk of this type, accessible from database servers on both networks, although normally only the primary network is actively using the disk. Should the primary fail, the standby can resume operations from the database image. To prevent the disk itself from becoming a single point of failure, a second "mirror" disk can be maintained, also accessible from both networks. This is typically implemented using a modified kernel disk driver, which writes duplicate logical blocks to independent controllers (lest loss of a controller become a single point of failure).

Initially we found this to be an attractive solution. It has one major advantage: the database image contains all committed transactions when the standby network takes control. (As we discussed previously, this is not true of the "periodic transfer" scheme, where it is possible to lose transactions occurring since the last transfer).

There are a number of disadvantages, however.

- i. *Potential for lengthy recovery time*. A hardware

or software bug in the primary side could write bad data on the disk. Before the standby database server can take control of the database, it must run a "database consistency check" (a DBMS equivalent to a UNIX `fck`). This can take a long time. If this check finds corruption, the database may prove to be unsalvageable, requiring restoring the entire database followed by a "roll-forward"—a playback of transaction logs saved in earlier checkpoints.

- ii. *Requirement for memory-cache write-through.* Though the DBMS guarantees that a committed transaction will be written to disk, it only writes it to the transaction log. Only when a database checkpoint is done (equivalent to a UNIX `sync`) are changed portions of the database image written out. Even if a crash does not garble the disk, the database image there will generally be incomplete, since cache contents are lost, requiring a (possibly lengthy) playback of the transaction log to recover the database image. One solution is to flush all updates immediately out to disk when a transaction is committed, so that the database image will be up-to-date in the event of a primary-side failure. This allows faster recovery at the cost of performance degradation during normal operation.
- iii. *Greater susceptibility to DBMS or operating system bugs.* The database's internal structures can be corrupted by bugs in the DBMS or operating-system software. We felt that the "periodic transfer" scheme offered better protection against this, since only "well-formed" transactions are sent to the standby side.

These disadvantages caused us to abandon this scheme with some regret, since it seemed easy to implement. It was clear, however, that lengthy outages were eminently possible. We felt that the "periodic transfer" approach offered us a better chance to meet our requirements for recovery time.

Cluster

A cluster configuration is another means of allowing multiple networks to share a common database image. Each network is connected to a machine belonging to the cluster. The database resides on a disk accessible from all members of the cluster. (The disk, of course, can be mirrored to prevent it from becoming a single point of failure). If the primary network fails, the standby network can resume operations from this copy of the database.

The VAX/VMS cluster, for example, provides all the advantages of the previously discussed dual-ported disk scheme, and has an interesting new capability as well. Instead of requiring that the standby network be quiescent, both networks can concurrently use the database. This simultaneous access is possible if the cluster-locking facility is used to resolve contention for database tables and records.

Two networks of trader workstations, each

connected to its own VAX could therefore concurrently access a common database. If one network goes down, users on that side switch to the other side. If the database is corrupted or otherwise affected by the failure, the DBMS software on the "healthy" side cleans up the situation (backs out failed transactions, removes dangling locks, etc.). This failover ability requires DBMS support (provided currently by ORACLE and RTI INGRES). There are clear advantages to having both networks active:

- i. *The "hot/hot" solution allows both sides to be active.* If one side goes down, half the users are unaffected.
- ii. *Both sides should normally enjoy good performance, since there is excess capacity to handle failover.*
- iii. *Failover is automatic as far as the database is concerned.* The surviving side cleans up the aborted transactions and unreleased locks. This saves time because the DBMS recovery can proceed in parallel with the switchover of users to the surviving side.
- iv. *Problems on either network are immediately detected.* In a "hot/hot" scheme, users continually exercise both network sides; a failure on one side is therefore immediately apparent. In a "hot/warm" scheme, pieces of the warm side could fail quietly (since no one is using them), and thus be unavailable when failing over from the "hot" side. Additional administrative and programming effort is required to avoid this (e.g., alternating sides on a daily basis and running "watchdog" programs or "dry-running" servers to detect warm-side failures).
- v. *This scheme does not lose any committed transactions.* Traders switching from the failed side will find all their completed transactions in the database. The cluster shares this advantage with the dual-ported disk configuration described earlier.

In the end, however, we found the cluster approach lackluster, since it has all the disadvantages of the dual-ported disk scheme as well. There is the possibility for a lengthy playback of the transaction log to recover a garbled database; there is also the problem of losing memory cache contents. There are some new disadvantages. Since DEC's ULTRIX has not been extended to use cluster locking, we would have had to use VMS. Introducing a second operating system would have obviously made network administration more complicated. Integration of VMS and UNIX applications could have been an even bigger headache. The current lack of a native DEC-supplied TCP/IP interface that would allow VMS to talk with UNIX machines also made us hesitate; we would have had to obtain this from a third-party vendor and this exceeded our comfort level.

Open issues/future directions

We feel that while the solution described in this paper meets our near-term requirements (and will be an interesting prototype) there are a number of long-term issues that we intend to address further.

- i. *Idle standby network.* Our present solution requires the standby network to be quiescent until there is a failure. We would naturally prefer to have both networks active, with half of the users on each side. The cluster solution (discussed above) could allow both networks to run "hot"—we may take another look at this when UNIX-based cluster products with TCP/IP network interfaces become available.
- ii. *Loss of transactions.* The "periodic transfer" scheme can lose a few seconds of committed transactions. This is not presently a concern since we have few transactions at risk, but it could become a problem if our transaction rate were to increase dramatically for future applications. A huge increase in traffic might be another reason for us to reconsider a cluster solution. We could also build tools that enable the traders to quickly identify and restore large numbers of missing transactions after a crash (these tools would be useful in any case to restore pending or "in-flight" transactions, which are a problem in any scheme).
- iii. *Incompatibility among vendor DBMS.* Defining a DBMS-based solution was made harder by the inability of different vendor DBMS to talk to each other. We therefore have a request: that DBMS vendors standardize on the data stream that flows between the database engine and the front end workstations, so that users can have mixed vendor front and back ends. We might eventually want, for example, to send SQL requests from a Sybase front end on a Sun workstation to a Tandem DBMS on a Tandem cluster. (Sybase is proposing its own standard. There is some ANSI activity in this area, X3H2's Remote Database Access - Service and Protocol).
It is difficult to control the whole picture when the data must be scattered among mainframes, variously flavored highly reliable machines, and traditional UNIX systems. No vendor is going to put its product on *every* machine that we might want to use. And in any case, one of the reasons for going with a UNIX LAN is to avoid becoming a vendor's captive.
- iv. *Component reliability.* As we said previously, we have not done much work to increase the reliability of individual components on our network, nor have we really considered or quantified the importance of MTTR (mean time to repair). We believe that our standby strategy coupled with ordinary hardware will result in high enough *aggregate* availability. The strategy does a good job of protecting against most

single points of failure². Still, suffering too frequent failures will make the system operationally painful. And once we transfer to the standby, the clock starts ticking until something fails there as well, before which the primary must be repaired and fully operational again. We're therefore thinking about using mirrored disks, or replacing our Sun dataservers with fault-tolerant and easily repairable machines (like Tandem or Sequoia) or machines which are not fault-tolerant but have a strong reputation for staying up (like Sequent).

- v. *Dynamic distributed recovery.* We also intend to improve availability by adding a limited form of dynamic software recovery (watchdogs, hot standby processes, etc.) in situations where this buys a lot and is easy to do.

As we said before, it would be nice to have a commercially available solution in which all networked machines could simultaneously do useful work, with a failure resulting in automatic reconfiguration and correct execution of the failed operation. In the future maybe this solution will fit with our IP-talking network nodes and still provide reliability which will be transparent to the application programmer. (We can always hope). While we're waiting we'll gain experience with our current solution.

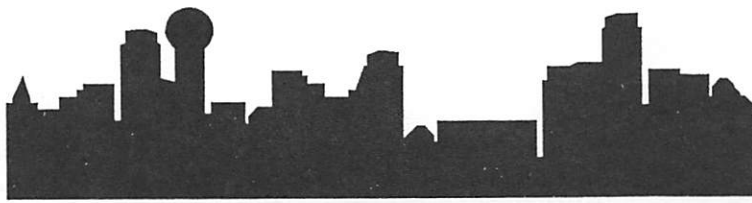
Acknowledgments

We would like to thank the people at Sybase, Inc.—in particular, Bob Epstein and Tom Haggin—for assistance in building the "periodic-transfer" prototype described in the paper. Hector Garcia-Molina and Dale Skeen evaluated our ideas at an early stage in our investigation, providing much useful input. Grace Hucko and Andy Van Hise helped to define our requirements for DBMS performance.

References

- [1] Stankovic, J. A., Ramamritham, K., and Kohler, W., "A Review of Current Research and Critical Issues in Distributed System Software," in *Concurrency Control and Reliability in Distributed Systems*, ed. B. K. Bhargava, Van Nostrand Reinhold Co., New York, 1987.
- [2] Sybase, Inc. *Database Administration Guide*, June 1987.
- [3] Garcia-Molina, H., "Reliability Issues for Fully Replicated Distributed Databases," *Computer*, Vol. 15, No. 9, pp.34-42, Sept. 1982.

²We can imagine types of non-transient system software bugs (or viruses) which could take out both networks, first the primary, and then the standby after the switchover. But none of the schemes that we considered is immune to this type of failure.



R. L. Murphy
Southwest Research Institute
Division 15
6220 Culebra Rd.
San Antonio, TX 78284

Comparing the Efficiency of the Internet Protocols to DECNET

ABSTRACT

It is necessary to transfer 50 megabytes per day from a computer on a wide area network (DECNET) to a computer on a local area network (TCP/IP). This operation must be done in a reasonable amount of "wall clock" time using a reasonable amount of system resources. A series of experiments are conducted to determine where bottlenecks occur and to test whether either protocol family is more robust than the other. Measurements of the actual throughput of the DECNET network, the TCP/IP network, and the associated gateway are presented and compared. Software tools under VMS, UNIX, and some written by the author are used and the results of these tests are focused towards recommendations for improving the performance of the gateway between the two networks.

Introduction

The goal of this study is to predict the actual data throughput of a data path that utilizes both DECnet and the DARPA Internet protocols. This hybrid network combines two of the most widely used network protocol groups and hardware technology that supports both Wide Area (WAN) and Local Area (LAN) networks. This configuration will be used for production work beginning in 1988. More specifically, we will examine the relative efficiency of each protocol and try to locate sources of overhead.

The study of protocol overhead is becoming increasingly important as new hardware technologies push the theoretical network speed upward. The gap between real performance and potential performance represents an area where protocol design and software have to be carefully considered. "We've done some benchmark tests on OSI," says Larry Green, director of technology at interfacemaker Advanced Computer Communications, "and we find that, at best, you can put data through it at about 2.2 Mbit/s. So when data comes down the network at 100 Mbit/s and hits the network and transport layers, it's just like hitting a brick wall."^[1] While the "brick wall" analogy is only partially applicable to the hybrid network involved in this study, the effect will be much the same if it takes longer to transfer data than to acquire it.

This study will not look at the areas of transmission error rates, network loading effects, or collision effects on CSMA/CD networks. These topics are beyond the scope of this effort.^[2]

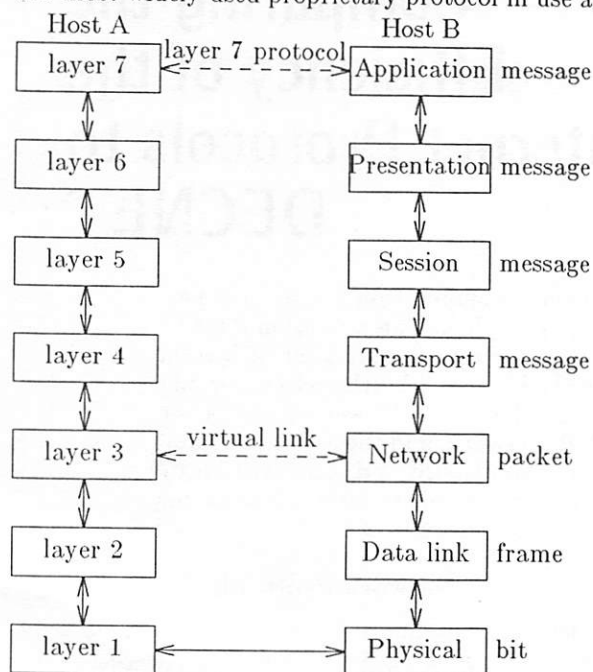
Network Protocols

Before examining the protocols studied here, it is worthwhile to look at the paradigm proposed by the International Standards Organization (ISO). The figure below illustrates several important concepts used to define and simplify network design. The communicating entities at a given level on each host are called *peer processes*. A process can be considered to communicate only with its peer at the same level. These processes have a *virtual path* across the network; only at level 1 does a physical path exist.^[3] Data from host A to host B actually passes down through each layer to reach the physical link level, whereupon it is transferred and then passed back up to the level of origination. The ISO nomenclature for the data passed between peers is the Protocol Data Unit (PDU), an entity commonly called a "packet". A particular level, such as the Transport level, utilizes Transport PDUs or TPDU's. The data passed between protocol levels is called a Service Data Unit (SDU). An examination of the SDU interface can reveal implementation complexities which result in overhead, something that does not show up when studying the protocol definition.^[4]

In practice many network implementations use hardware to do the work at levels one and two. Typically this results in greater throughput than pure software implementations. The hardware may be a single "chip" or an entire board containing a microprocessor, memory for buffers, and so forth.

DECnet

In this paper the term "DECnet" will be used to refer to a particular implementation of the Digital Network Architecture (DNA). DECnet is probably the most widely used proprietary protocol in use after



ISO Layered Model

IBM's SNA. DECnet encompasses all seven levels of the ISO model, but as the table below shows, there is not a one-to-one correspondence between the ISO network levels and those in DECnet.

Level	ISO	DECnet
7	Application	Application
6	Presentation	
5	Session	Not used
4	Transport	Network services
3	Network	Transport
2	Data link	DDCMP
1	Physical	Physical

The exact implementation of the higher levels of DECnet is not easily discerned because it is proprietary and these levels are deeply buried into the operating system. At the application level, file access is provided across the network in a transparent fashion, a feature that is not provided by the Internet

protocols. At the data link level we will utilize the Digital Data Communications Message Protocol (DDCMP)^[5] as well as Ethernet. DDCMP was designed to be easy to implement on many types of communications channels, so its basic data unit is a character (8 bits) rather than a bit. The DDCMP frame format is shown in the next figure. With only 14 bits in the "data count" field, messages longer than 16384 bytes can not pass atomically through the data link level.

The Internet Protocols

The Advanced Research Projects Agency (ARPA) funded the specification and development of an extensive set of protocols. The first large scale research network, ARPANET, was an outgrowth of this research. These protocols are widely used because they are well defined and non-proprietary; many different types of computers communicate via the Internet protocols. These protocols have become intimately related to UNIX because they are supported (a subset) by the Berkeley version of UNIX.

There are three protocols which concern this study: the *Internet Protocol (IP)*, the *User Datagram Protocol (UDP)*, and the *Transmission Control Protocol (TCP)*.^[6] The acronym TCP/IP is generally used as a catch-all expression for software that spans all seven levels of the ISO model.

The Internet Protocol is used to send data between hosts. It is not concerned with routing, reliability of delivery, and task-to-task delivery. The UDP handles message delivery between communicating tasks on different hosts and message fragmentation and demultiplexing. Of course, demultiplexing and reassembling messages lowers the efficiency of the data transfer. The use of *datagrams* is considered an unreliable method of delivery because there is no acknowledgement of delivery and the order of delivery may not be preserved. This is called a *connectionless* mode of operation. This means that there is no virtual circuit or link established between two communicating entities.

By contrast, TCP is *connection* oriented. Historically, it was designed to provide reliable message delivery over the sometimes poor circuits used by the ARPANET. The overhead involved in this will be described later in this paper.

The computers used by the author all use Ethernet at the data link level when running the Internet protocols. Ethernet is an example of a Carrier Sense Multiple Access with Collision Detection (CSMA/CD) scheme for maximizing bandwidth utilization. It uses

syn	syn	class	count	flag	response	sequence	address	header	data	data
			14 bits	2 bits	8 bits	8 bits	8 bits	CRC	8 bit	CRC
								16 bits	characters	16 bits

DDCMP Format

Manchester coding to provide a built-in clock and this is treated like a carrier, even though the transmission is baseband. Because Manchester coding requires a state transition in the middle of every bit, a 10 Mbit/s Ethernet effectively has a 20 MHz clock. Each station listens as it transmits so it can immediately detect collisions and cease the transmission. We are concerned mainly with the overhead introduced by the Internet protocol layers above the Ethernet and to a lesser extent that of the Ethernet itself.

Testbed description

The computer systems used in the tests are a Convex XP-1, two MicroVAX IIs, a Masscomp MC500, a VAX 11/750, and a VAX 8800 with a router/server. The Convex, Masscomp, and MicroVAX all have the EXOS 201 Ethernet board made by Excelan. This board is capable of running UDP and TCP on its internal processor, but this is not done on the XP-1. The XP-1 has a separate I/O processor that handles memory-to-interface transfers.

DECnet is supported by a 9600 baud asynchronous link between a DZQ11 on one MicroVAX and a DMF32 on the 11/750. The DMF32 is a DMA type device, while the DZQ11 is a line multiplexor. The 11/750 is the source of the data and the XP-1 is the destination. The second DECnet link is a 9600 baud synchronous line between the router/server and a DMV11 on the second MicroVAX.

Although the real application involves copying files across a network, file transfers are not explored in depth because of uncertainties introduced by variance in disk performance. For instance, the Convex uses an advanced file I/O management technique called "disk striping" that provides high data rates. At the other extreme, the Masscomp has three disk drives on a single controller, resulting in relatively sluggish performance. Some file copying data will be presented for comparison, but it is difficult to remove file system/disk performance effects from the measurements.^[7]

Test Results

Before describing the tests that were performed and the results obtained, it is informative to look at some background data. The table below contains

DECnet/Ethernet Mbit/s		
Hardware interface	file copy	task-to-task
DEUNA (unibus)	.5	1.4
DELUA (unibus)	.6	4.1
DEBNT (BI bus)		3.2
DEQNA (qbus)	.3	3.2

results of an independent test of DECnet data throughput¹.

¹From a networking seminar given by DEC on March 23, 1987.

The bandwidth utilization is 32% in the case of the DEQNA task-to-task test, assuming a 10 Mbit/s Ethernet. This unusual value turns out to be quite different than those measured in the DECnet tests described later, whereas the file copy results are very close. Unfortunately, no details of the test conditions were provided, so a more direct comparison is not possible. Information on the size of the buffers transferred, the nature of the communicating tasks, and the conditions under which the tests were performed would help to evaluate the results.

The Internet protocols were tested using C programs because the socket interface structures are easier to use in C. FORTRAN was the test program language for the DECnet tests. In both cases, the operating system provides a system call that returns "wall clock" time with a resolution of 0.01 second. An independent test was run to place an upper bound on the overhead of reading this clock value; it seems to be less than 310 μ sec. This value is unlikely to skew the test results as they are measuring much larger durations.

All tests were done when the network was idle and the host computers were lightly loaded. All of the machines running TCP/IP use *trailer* encapsulation to minimize buffer copying.^[4] Also, none of these systems were running *routed*, the BSD Routing Information Protocol (RIP) server, although it is supported. In all cases the tests were repeated so that the results could be averaged to smooth the effect of system load changes. No effort was made to remove the UNIX system daemons; it is assumed again that the effect of daemons waking up in the middle of a test will be averaged out. The tables and graphs which follow illustrate some results.

Seconds	Bytes	kbytes/sec
.0125	256	20.48
.0125	512	40.96
.0125	768	61.44
.013	1024	78.77
.014	1280	91.43
.018	1536	85.33
.020	1792	89.60
.021	2048	97.52

Masscomp to Convex UDP test

This test was intended to provide measurements of UDP performance for buffer sizes up to 16384 bytes, but the Masscomp system has a buffer limitation of 2048 bytes². The UDP/IP definition allows almost 64K bytes to be passed, so 2048 bytes seems

²This is because the 201 board is running TCP/IP and has little memory left for buffers. The latest release of UNIX for this system allows TCP/IP to be run in the kernel, thus solving this problem at the expense of speed.

unreasonably low. The last value in the table represents a bandwidth utilization of 7.8%, a result which is consistent with the TCP/IP study done by Miya.^[7] The data in this table are plotted in figure 1.

Seconds	Bytes	kbytes/sec
3.96	500	.128
5.61	1000	.178
15.48	5000	.323

Asynchronous DECnet task-to-task

Given a 9600 baud asynchronous line, the bandwidth utilization is approximately 33.4% for the largest buffer size. However, over the synchronous link the utilization is close to 88%. When Ethernet is used as the DECnet link layer, the utilization ranges between 2.6% and 3.58%. So it seems that the link level protocol is a critical factor in determining throughput.

Sources of protocol overhead

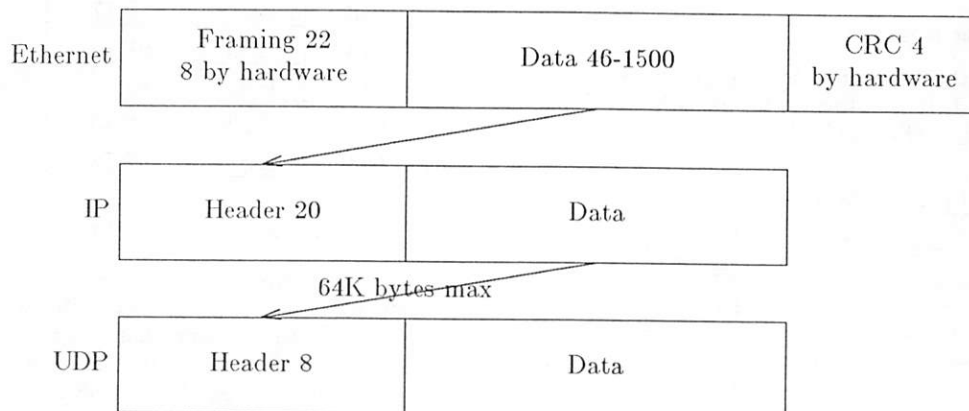
It is evident that under certain circumstances DECnet can utilize the majority of the bandwidth of the physical communications channel. Unfortunately, no equivalent "data pipe" could be used for the TCP/IP link layer, so a similar performance can only be projected. To understand why this is so, both the details of the protocol and the SDU interfaces must be uncovered. Some of the complexities both protocols must handle are:

- The handling of multiple outstanding (unacknowledged) frames. Both protocols have equivalent features.
- CRC and checksum generation and checking. There can be a CRC and several checksums, each used by a different layer. This seems to be significant in at least one TCP/IP implementation.^[7] Here is one are where DECnet appears to have less overhead.

- Message fragmentation and reassembly. IP is simpler than DECnet, unless the link layer is DDCMP.
- The use of control messages. Some control functions may be "piggybacked" on data messages, thus saving time. A protocol may use implicit negative acknowledgements (NAKs), which is how TCP works. This requires a time-out function and this wait time is overhead. Both protocols allow one acknowledgement to cover multiple PDUs, but the plethora of DECnet control functions definitely complicates the implementation.
- Complicated routing algorithms may be used. Routing delays come from searching routing tables and address verification. This topic is too involved to be considered here.
- Connection establishment. Short packets are more efficiently sent over a connection oriented link because the inclusion of an address with each block of data would waste bandwidth. On the other hand, infrequent transfers are less efficient because of the overhead involved in establishing the connection. Here DECnet is more complicated than TCP/IP and its operating system dependencies become obvious.

Because the higher network layers are not involved in the Internet tests and they are in DECnet, this is an obvious source of overhead. There is no easy way to address the lower protocol layers in DECnet as there are in the BSD 4.x implementations of TCP/IP. Even task-to-task communication uses the presentation layer and all links are connection oriented.

The next diagram illustrates the fragmentation overhead for UDP/IP on top of Ethernet. The formula predicts the idealized bandwidth, given no software or hardware overhead, and it is plotted in figure 1. There is a very big gap between this curve and that derived by the test software. This would seem to imply that UDP/IP is not inherently inefficient, but perhaps the implementations are. If



64K bytes max
Fragmentation and Framing Overhead for UDP

one compares the UDP/IP results with Miya's TCP/IP results, the expected difference is seen.

For the above diagram, we can approximate the actual channel efficiency β with the equation below:

$$\beta = \frac{d}{d + f}$$

$$f = \begin{cases} 54 + (18 - d) & \text{if } d \leq 18 \\ 54 & \text{if } 18 < d \leq 1472 \\ 54 + 26np & \text{if } 1472 < d \leq 16384 \end{cases}$$

In this equation d is the number of data bytes and

$$np = INT \left(\frac{d - 1472}{1500} \right)$$

Conclusions

Under the assumption of 88% efficiency and no file system overhead, it will take about 2.25 hours to move 50 megabytes of data across a hybrid DECnet-TCP/IP network. The DECnet physical link will be capable of 56 Kbit/s and the TCP/IP side will use Ethernet. Tests using the Internet File Transfer Protocol (FTP) with the DECnet copy operation have yielded an aggregate data rate of 0.424 Kbyte/s. Although this rate includes disk I/O overhead, it should be achievable under normal loading. Unfortunately, at this rate, it will require about 32 hours to transfer the 50 megabytes and this is not acceptable.

One can conclude that the studied Internet protocols are simpler than the 7 layer DECnet, hence they are slightly more robust given parity at the link layer. However, it is the data link layer that effects this application the most. It appears that implementations are a primary issue as well. For instance, in this application there is a definite data source and

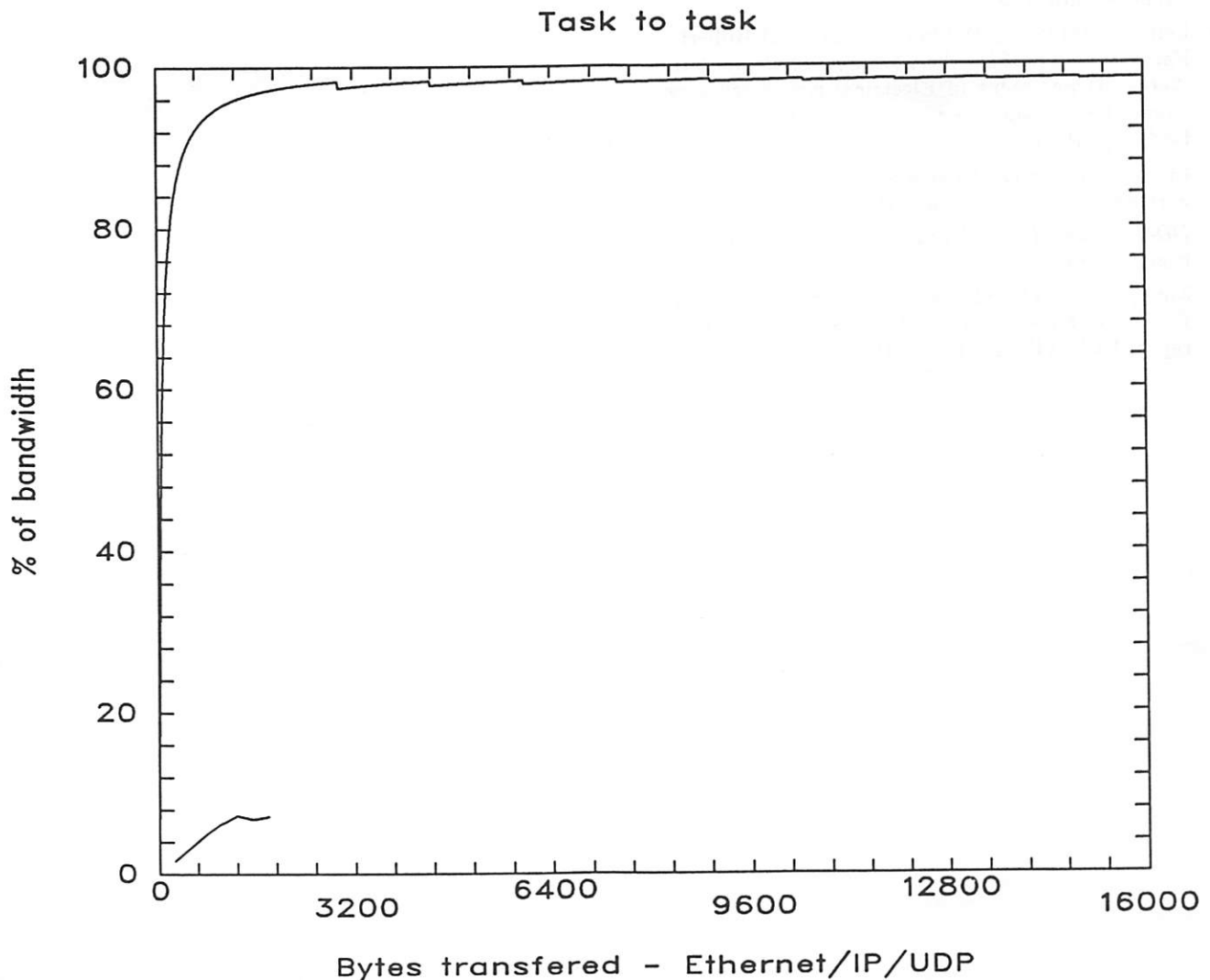


Figure 1

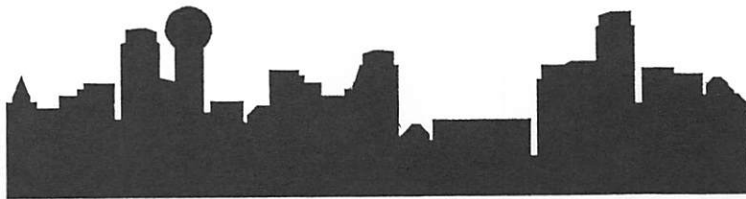
consumer. Over 95% of all the data flows in one direction. If the Ethernet board equally allocates the on-board buffers for receiving and transmitting, then both the source and consumer are wasting a resource. Allocating asymmetric buffers might allow larger packets to be passed to the link layer, where microprocessors do the work. The next phase of this study will explore this scheme and others as possible means to improve throughput.

Acknowledgements

The author appreciates the valuable assistance of Sandee Jeffers in collecting some of the data used here.

References

1. *Data Communications*. Vol. 15, No. 4, April 1987, p.15.
2. Arthurs, E. and B.W. Stuck. *A Computer and Communications Network Performance Analysis Primer*. Prentice-Hall, 1985.
3. Tanenbaum, Andrew S. *Computer Networks*. Prentice-Hall, 1981.
4. Leffler, Samuel J., William N. Joy, and Robert S. Fabry. . 4.2 *BSD Networking Implementation Notes*. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1983.
5. Friend, George E. *Understanding Data Communications*. Howard W. Sams, 1984.
6. *DDN Protocol Handbook*. Vol. II, SRI International, 1985.
7. Miya, E.N. *Network Performance Studies on 4BSD UNIX Systems*. NASA Ames Research Center, report ECT TR 84-3-ENM, 1984.



USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

PMON: Graphical Performance Monitoring Tool

Paul Jatkowski
Rich Inc.,
3747 Acorn Drive,
Franklin Park, IL 60131
ihnp4!richp1!pej

Mike Akre
AT&T
4513 Western Ave
Lisle, IL 60532
ihnp4!cuuxb!akre

ABSTRACT

PMON is an interactive performance monitoring tool that presents graphical displays of the current state of a running system. It runs on UNIX System V on the AT&T 3B computer line, although the concept and design are extensible to any System V implementation. *Sar(1)* type data is presented in a format that is easy for system administrators to use for system performance analysis and tuning. *PMON* uses *curses* for its terminal interface, making it usable on a wide range of terminals. *PMON*, although unofficial and unsupported, has become ubiquitous within AT&T.

This paper describes both *PMON*'s user interface and internals. It includes the authors' feelings as to why the presentation of the data is critical to the usefulness of the tool. The basic mechanism used for gathering system performance information will be discussed.

Introduction

PMON is a program that displays system activity data graphically on the user's terminal. It displays the data in "real time" giving the user a feeling for what the machine is doing at a particular instant in time. The user can set the time between updates to allow a choice between rapid (but possibly expensive) or slower updates. This paper will briefly describe *PMON*'s history and current status. Its user interface and internal operation will be described in some detail.

Work on *PMON* began in early 1984. By October of 1984 it was necessary to put the source under SCCS control to keep the various versions straight. The initial work was motivated by trying to understand how the kernel kept track of its activity, as well as trying to find a better way to present the data than *sar* provided. Two different versions of *PMON* currently exist: one for swapping versions of System V and another for paging versions of System V. All of the recent effort has been in the paging version, which is currently about 4400 lines of C code. (This work was done by Paul Jatkowski while he was with AT&T.)

As stated above, there is one version of *PMON* for paging versions of UNIX System V. The source

contains *#ifdef*'s for differences between different ports, but by and large these differences are small. The code is currently running on UNIX System V Release 3.1 on the AT&T 3B2, 3B15, 3B20 and the AT&T 6386 WGS machines. The latest port, to the 6386 WGS (an Intel 80386 based machine) simply required recompilation of the source. The major porting problems tend to be differences in system structures (header files) or flag definitions. These differences seem to grow smaller with each new release.

The source for *PMON* is available from the AT&T UNIX System Toolchest.

User Interface

Many system administrators have trouble using *sar(1)* data. Even though *PMON* presents basically the same data, people seem to find it more understandable than row after row of numbers. *PMON* has not had its own documentation, so users are generally referred to standard documentation on *sar* and system tuning.

With *PMON* the user can watch a screen and look for the bars that are the longest. This makes finding parameters that are currently being "stressed" much easier than with *sar*. *PMON* also

displays the percentage of "the maximum seen" in many cases. This type of display shows the current usage level compared to the maximum seen since the program was invoked. This feature gives the user a feeling for relative performance. Many users have reported that *PMON* is much easier to use than *sar*, even though both programs work with the same data.

PMON displays two basic types of bar graphs. The first type is an "absolute" bar graph, where the values being displayed have known upper limits. The bar is shown as a percentage of the possible maximum. An example of this type of display would be the percentage of time the CPU is in user mode, kernel mode, or idle, or the amount of swap space currently allocated. There are many parameters that do not have fixed upper bounds, for example, the number of disk reads or system calls per second. *PMON* uses a "percentage of maximum seen" graph for these types of displays. In other words, the display will show the current activity level as a percentage of the maximum recorded level for the current run. Most items are normalized to a per-second basis. This normalization allows the numbers to be compared even if *PMON* is sampling at different intervals.

PMON is totally menu driven with most commands requiring one or two key strokes. When the program is invoked with no command line arguments, the user is presented with a menu showing the possible choices (see Figure 1).

The options at this level allow the user to alter the sampling interval and select one of several top level pages which may have several sub pages. The sub pages are currently grouped into three categories:

- activity display – displays relating to *sar* tunable

variables

- system table display – displays relating to internal system tables, such as the *proc*, *inode*, and *region* tables.
- tunable parameter display – displays showing how much of certain tunable parameters are actually being used in the system.

The data for all sub pages in a group is updated when one of the sub pages is being displayed.

PMON also includes a file system display that shows the free inodes and blocks on mounted file systems. This display doesn't really fit into the rest of *PMON* and may be removed in the future.

Each of the menus and sub-menus will be presented with some example pages from each sub-menu.

Activity Display

The activity displays show the same data presented by *sar*. It currently has 10 sub pages including the initial help page. The initial help display for the activity screens is as follows:

```
minnie (1203) update=5  System Activity
(Page 0)                Sat Dec  5 11:21:42 1987
```

page	display
0	This menu
1	cpu split information
2	block io subsystem
3	character/raw io
4	scheduler counts
5	paging info
6	more paging info(%s)
7	remote services info

```
performance monitor (version 3B2(3.1) 1009.1)  Sat Dec  5 11:17:49 1987
options:
```

```

s/d      set update delay
f        select file system display
a        select activity display (sub menu)
T        select system table display (sub menu)
t        select tunable parameter display (sub menu)
+/-      increment/decrement sub page
c        redraw screen
q        terminate program
!        escape to sub shell
?        display this menu
```

Select an option to continue

Note: there may be up to a sample time delay
before a selected option takes effect

Copyright (c) 1984, 1987 AT&T
Comments should be sent to {ihnp4!}cuuxb!ucsmail

Figure 1

8 rfs cache info
9 streams activity info

- CPU Split Information - breakdown of where the processor is spending its time; User mode, Kernel mode, Idle, or Waiting for I/O to complete. Some kernel implementations also keep track of when the processor is idle waiting for memory to free up.

This page shows information for the current sample and totals since system boot (although if the system has been up for long enough, the counters wrap around and make the "since boot" information invalid). Figure 2 depicts these graphs which have absolute ranges of 0% to 100% for each category.

- Block I/O Subsystem - This page displays

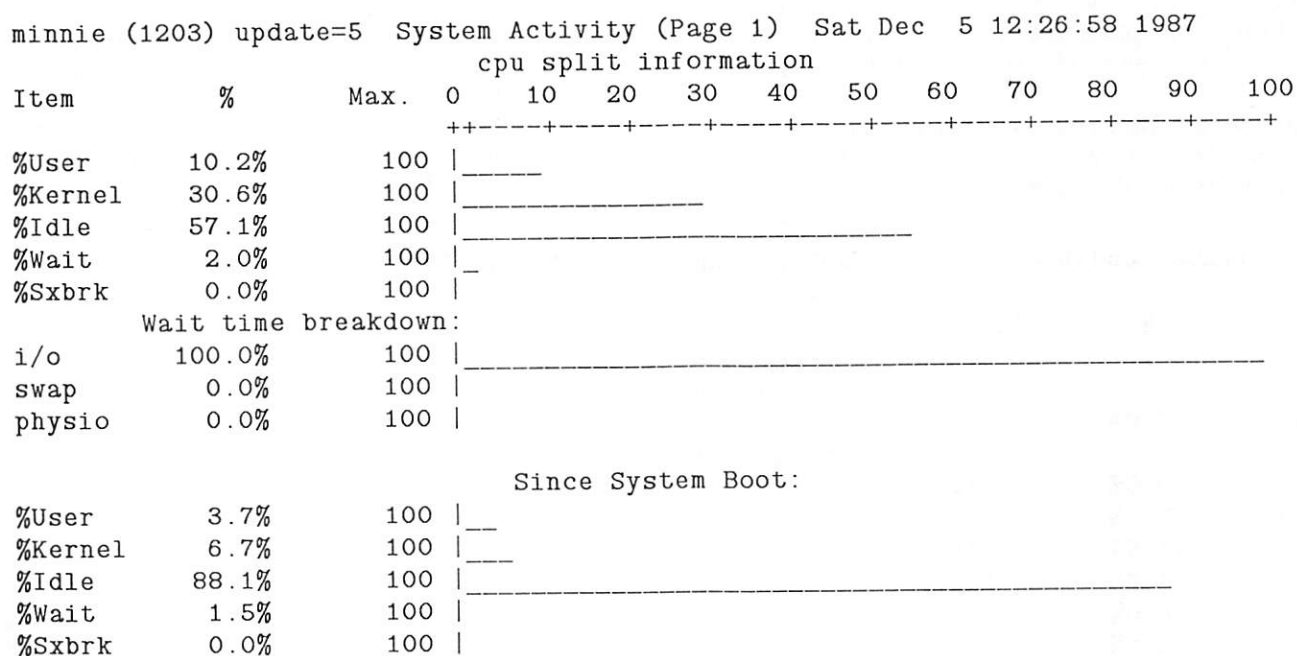


Figure 2

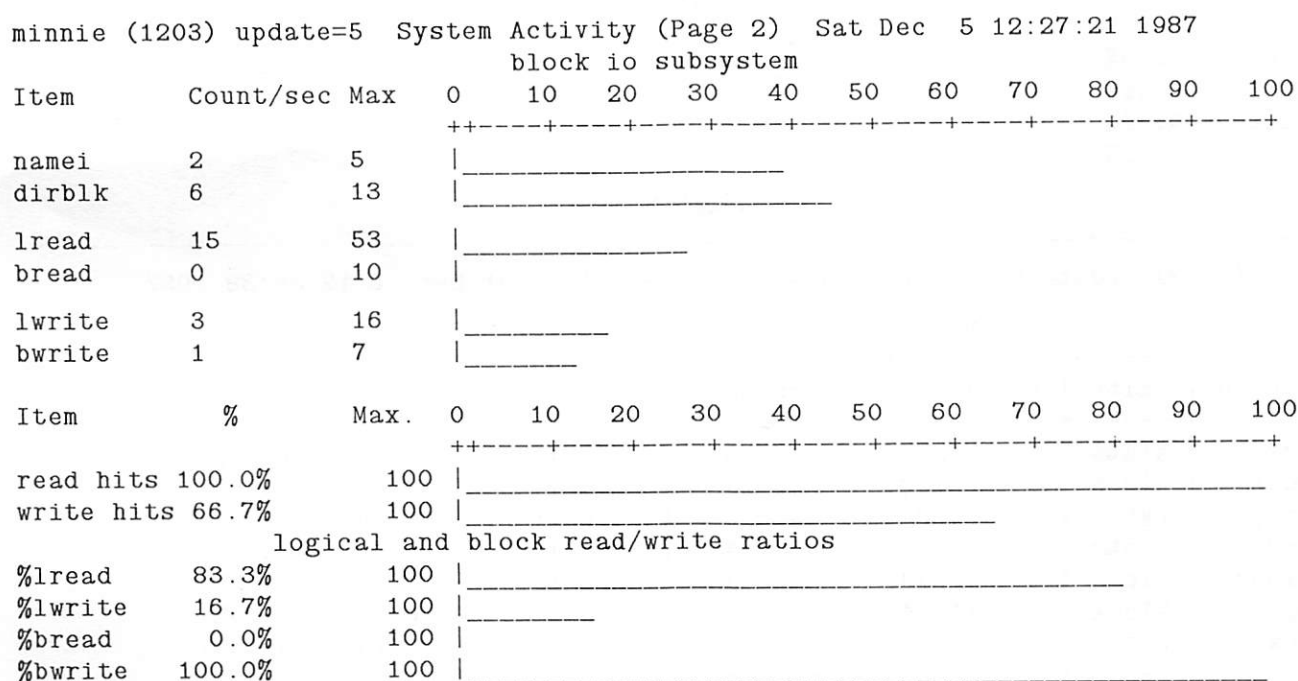


Figure 3

information about the buffer pool. It shows the number of calls to *namei* and *dirblk* per second as well as the number of logical reads, block reads, logical writes, and block writes per second. These items are displayed as a percentage of maximum seen.

This page includes the read and write cache hit ratio (percentage of reads/writes that were satisfied from the buffer cache), and the read/write ratios for logical and block reads. These last items are displayed on absolute 0% to 100% graphs in Figure 3.

- Character and Raw I/O – This page displays information relating to raw and character I/O. All of these items are displayed in the “percentage of

maximum” type display. The values displayed include physical reads, physical writes, blocks swapped in, blocks swapped out, number of characters read and written by processes, terminal receive interrupts, transmit interrupts and modem interrupts per second. The number of characters per second read or written via the tty raw queue, canonical queue, and output queues are also displayed.

- Scheduler Counts – This page contains counts generally relating to scheduling and process switching activity. It displays the number of process switches, number of processes in the run and swap queues, number of forks, execs, system calls, swap in and out operations per second.

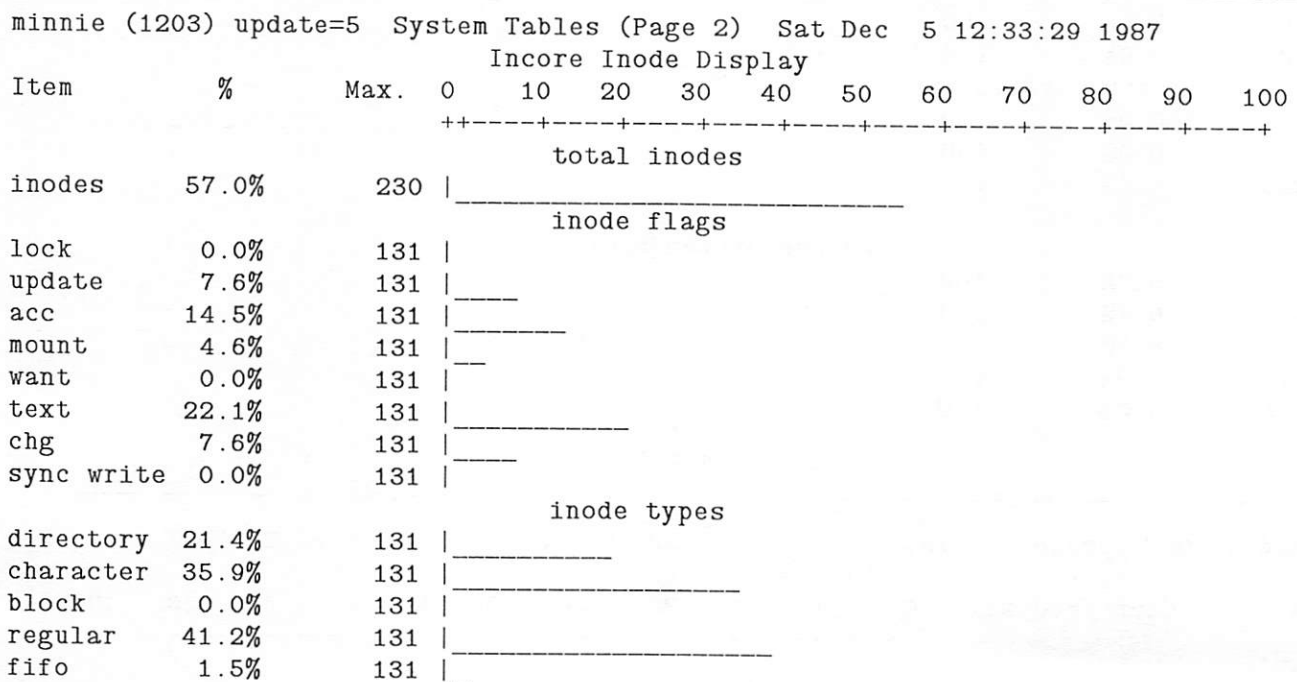


Figure 4

minnie (1203) update=5 Tunable Parameter (Page 1) Sat Dec 5 12:34:38 1987

Tunable parameters - table format

Resource	units	total	allocated/used	free
Files	slots	256	97 (37.9%)	159 (62.1%)
Inodes	slots	230	132 (57.4%)	98 (42.6%)
Memory	bytes	4194304	3932160 (93.8%)	262144 (6.3%)
Procs	slots	130	46 (35.4%)	84 (64.6%)
Regions	slots	340	132 (38.8%)	208 (61.2%)
Swap	blocks	11984	1076 (9.0%)	10908 (91.0%)
Flocks	cnt	16	3 (18.8%)	13 (81.3%)

Figure 5

- **Paging Information** – This page contains counts related to the paging subsystem. The number of protection faults per second is displayed and broken down into *copy on write* and *page steals*. The number of virtual faults per second is displayed and broken down into *demand fill*, *swap fill*, *cache fill*, and *file fill*. Also displayed are the number of pages freed per second (by system memory reclaiming) and the number of pages found unmodified both on the swap device or a file system.
- **More Paging Information (percentages)** – This page displays the information from the previous page in a different format. The ratio of *copy on write* and *page steal* faults are displayed for both the current sample and since system boot. The ratio of *demand fill*, *swap fill*, *cache fill*, and *file fill* faults are displayed for both the current sample and since system boot.
- **Remote Services Information** – This page displays information relating to Remote File Sharing (RFS) activity, such as incoming and outgoing requests for system calls, reads, writes, and execs. The number of characters read and written per second for both incoming and outgoing requests is also displayed. This page also shows the number of active servers and the server queue activity.
- **RFS Caching Information** – RFS in UNIX System V Release 3.1 implements client caching to help reduce network traffic. This page displays cache hit and cache disable activity and the number of RFS messages sent and received per second.

- **STREAMS Activity Information** – The last page of the activity display pages shows information about STREAMS buffers. It displays the number of STREAMS message and data blocks allocated per second and includes a breakdown of the allocations per second for each data block size.

System Table Display

The system table display summarizes information about major system internal data structures. This information is not available from *sar*. Six sub pages including the initial help page exist. The initial help display for the system table display is as follows:

```
minnie (1203) update=5  System Tables
(Page 0)  Sat Dec  5 12:32:36 1987
```

page	display
0	This menu
1	process state
2	inode state
3	file state
4	system region state
5	swap table

- The process state display has one bar that displays the number of active processes as a percentage of the maximum possible processes. All of the rest of the bars are displayed as a percentage of active processes. The state of the active processes (*sleep*, *run*, *zombie*, *grow*, *other*) is displayed as well as the percentage of processes loaded in memory, swapped out and locked in memory.
- The inode state display shows information about

```
minnie (1203) update=5  Tunable Parameter (Page 3)  Sat Dec  5 12:35:25 1987
```

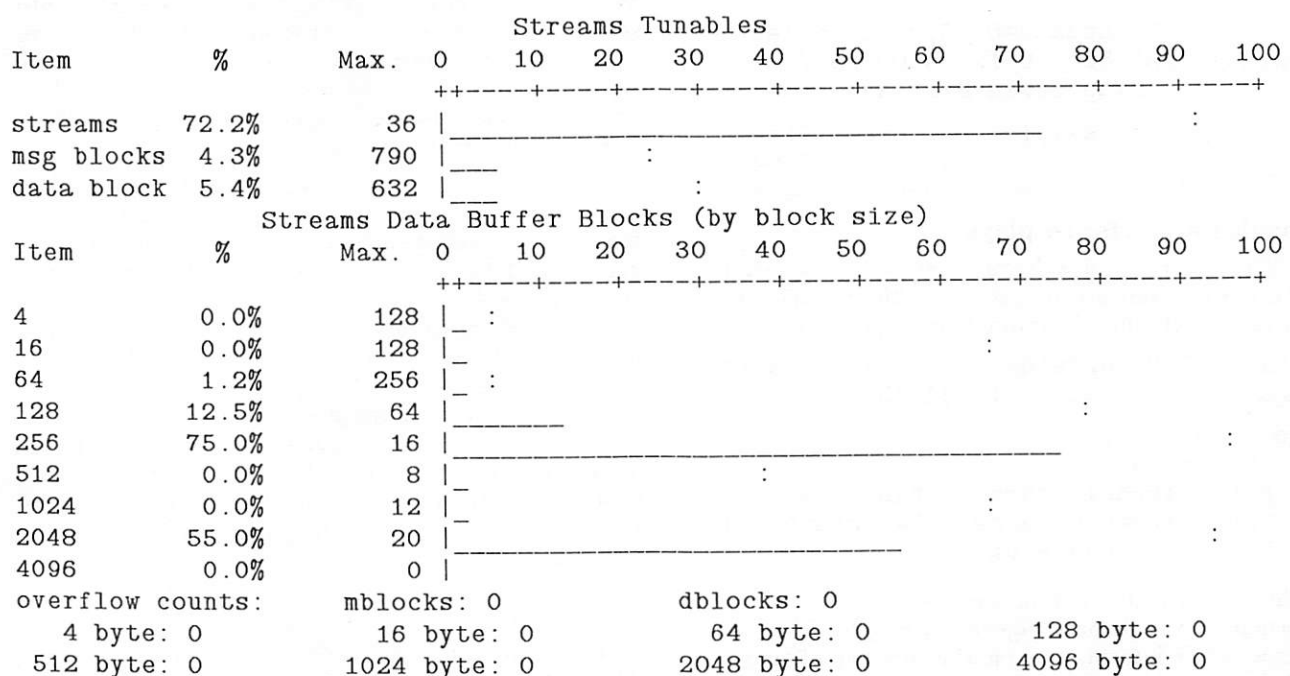


Figure 6

the incore inode table. One bar displays the number of active inodes as a percentage of the maximum possible. The rest of the bar graphs display as a percentage of the currently active inodes. The first set of bars display inode flags: *locked*, *update*, *acc*, *mount*, *want*, *text*, *chg*, and *sync write*. The second set of bars display the type of inodes; directory, character special, block special, regular file, or fifo. Figure 4 depicts an example page.

- The file state table shows information about the incore file table. One bar shows the number of active file table entries as a percentage of the maximum possible. The rest of the bars are displayed as a percentage of the number of currently active entries. The bars display the modes in the file table entries: *read*, *write*, *ndelay*, *append*, and *synchrous*.
- The system region table display shows information relating to the global region table. One bar shows the number of active regions as a percentage of the maximum possible. The rest of the bars are displayed as a percentage of the number of regions currently active. The first set of bars displays the types of regions: *private*, *shared text*, or *shared memory*. The second set of bars shows the region states: *nofree*, *done*, *noshare*, *wanted*, and *waiting*.
- The swap table display shows information relating to the system swap or paging areas on disk. It is a tabular display rather than a bar graph display, and shows the major and minor number of each swap device, the starting block for swap on the device, the number of blocks allocated and the number of blocks still free. An example of this page is as follows:

```
minnie (1203) update=5  System Tables
(Page 5)  Sat Dec  5 12:34:09 1987
          Swap Table Display

major  minor  swaplo  blocks  free
  17      1      0    4048   3504
  17     17      0    7936   7404
```

Tunable Parameter Displays

This final set of sub pages shows information relating to administrator set tunable parameters. Four pages including the initial help display exist:

```
minnie (1203) update=5  Tunable Parameter
(Page 0)  Sat Dec  5 12:34:33 1987
```

```
page    Display
0  This menu
1  Basic Tunable parms - table fmt
2  Basic Tunable parms - bar graph fmt
3  Streams Tunable parms
```

- The basic tunable parameters are available in two formats – a table or a page of bar graphs. The first page shows information on the number of active files, incore inodes, process table slots, system regions, and file and record locks. The amount of physical memory and swap space currently

allocated is also displayed. The table format is seen in Figure 5.

- The basic tunable parameter bar graph display shows the same items as the previous display, except in bar graph format.
- The STREAMS tunables page is a relatively new addition to *PMON*. Since this page displays usage of a fixed resource, the displays show the percentage of the maximum allowable values. A marker character has been added to show the maximum value seen for each item. The first set of bars displays the number of streams, message blocks, and data blocks currently allocated. The second set of bars break down the allocation of data blocks by block size. The last section of the page displays overflow counts for message blocks, data blocks, and again, a breakdown by data block size. An example of this page is seen in Figure 6.

Program Basics

This section describes some of the basic concepts and building blocks that make writing a program such as *PMON* both possible and easy to manage. A little background on how the UNIX System V kernel keeps statistics and how to retrieve them will be presented. Also, a suggested set of basic routines will be described.

System Statistics

The UNIX System typically keeps its statistics in the form of counters. Each time a particular event occurs a counter is incremented. If the counter is examined at a known time interval, the difference can be used to calculate a per-second rate for the counter. Most of the counters tend to be grouped together into structures. The major structures are defined in */usr/include/sys/sysinfo.h*. This file defines the *sysinfo* structure and possibly a few others, depending on the particular UNIX System release. *Sar* uses these structures for its information.

UNIX System V does not provide any system calls to get direct access to the various counter structures. Even without such an interface, it's not hard to get to the data. The first step is to find out where the structure resides in memory. To do this, the structure must be looked up in the symbol table (or name list) for */unix*. The *nlist(3)* library function has been provided to do just this. *Nlist(3)* is called with the *a.out* file name and a partially filled in *nlist* structure (with possibly several symbol names). It will fill in the address or value of each symbol in the input structure that it finds in the *a.out* symbol table. In the case of the kernel, the value is the virtual address of the structure in kernel memory.

Once the address of the structure is known, it has to be read from kernel memory. Two special files deal with machine memory: */dev/mem* and */dev/kmem*. The first allows programs to access physical memory with no address translation. While this special file is useful for some purposes, it is not what we need here.

`/dev/kmem`, on the other hand, allows access by kernel virtual address. To read the structure the program will need read permission on `/dev/kmem`. This permission is not normally allowed because of possible security problems (*PMON* avoids the potential security problems by being set up as a `setuid` program). Here is a small example that reads the `sysinfo` structure on a running system (error checking has been omitted for clarity):

```
#include <sys/types.h>
#include <sys/sysinfo.h>
#include <fcntl.h>
#include <a.out.h>
#undef n_name

struct nlist nl[] = {
    { "sysinfo" },
    { 0 }
};

#define SYSINFO nl[0].n_value

main()
{
    struct sysinfo my_sysinfo;
    int mem_fd;

    nlist("/unix",nl);
    printf("sysinfo structure loc 0x%x\n",
           SYSINFO);
    mem_fd = open("/dev/kmem",O_RDONLY);
    lseek(mem_fd,SYSINFO,0L);
    read(mem_fd,&my_sysinfo,sizeof(my_sysinfo));
    printf("bread=%d\n",my_sysinfo.bread);
}
```

Basic Routines

PMON uses several utility routines to help isolate the higher level tasks from the low level utility tasks. Among the low level utility routines is the *curves* library. It takes care of the terminal-dependent details of screen handling and newer releases even provide support for graphic line drawing characters on terminals.

Another group of basic routines deal with drawing bar graphs and headers for the graphs. For each type of bar graph, one routine draws it and another that draws the header line for that type of bar graph. These routines provide for consistent looking bar graphs on all of the displays and also make it easy to show additional or different information.

The previous example shows that getting structure data from memory requires two steps: a seek and a read. Adding a function or macro that checks each step for errors makes the main program cleaner, and warns of unexpected errors that would be hard to track down otherwise. Additionally *PMON* keeps track of the maximum, previous, and current delta values for many pieces of information (structure members). The code is much cleaner and easier to read when macros are defined to process the structure members. An example of such a macro might be:

```
#define UPDATE(x) \
    delta.x = current.x - previous.x; \
    if (delta.x > maxdelta.x) \
        maxdelta.x = delta.x; \
    previous.x = current.x;
```

Then the code would look like:

```
struct sysinfo current, previous,
i                maxdelta,delta;

...
UPDATE(bread)
UPDATE(lread)
...
```

By defining a good set of low level routines, it becomes very easy to add additional displays.

Program Flow

After the user specifies a page to display, *PMON* loops until it receives another command. It reads the relevant kernel counters, updates its own counters and data, prints the current values, and then waits until the next update.

Impact on the System

PMON, like any other program that runs on the system that it is observing, uses some system resources to do its own work. This overhead, although measurable, is normally quite small. The screen update interval determines the CPU usage: on an AT&T 3B2/600, *PMON* uses about 1.7% of the CPU when using the default update interval of five seconds. An interval of two seconds raises CPU usage to 5%, and an interval of zero is supported but not particularly useful.

Possible Future Directions

Programs like *PMON* are never completed; someone always wants it to be able to do something new. A few items that are currently under consideration include:

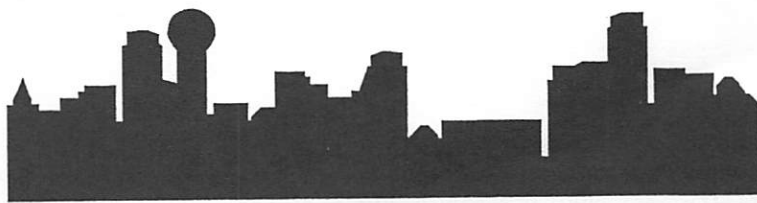
- Convert the floating point math to artificial precision integer calculations. *PMON* does a fair amount of calculations per update cycle; eliminating or reducing floating point operations might have a significant impact on some machines.
- Implement the statistics collecting in a device driver. This would eliminate problems with "non atomic" data collection. This enhancement has the disadvantage of somewhat reducing portability.
- Implement a logging/playback facility allowing data to be collected for later review.
- Add support for color terminals.
- One idea that has recently been implemented is to cache the information obtained from the *nlist(3)* function. *Nlist(3)* is a relatively expensive function. *PMON* saves the information in the file `/tmp/pmon.addr` so that after the first invocation it starts much quicker.

Conclusion

PMON has evolved over several years of use, both locally and throughout parts of AT&T. It provides functionality not provided by other performance monitoring tools. While it was originally developed in an effort to learn more about the internals of the UNIX System and its performance statistics, it has become an extremely useful tool for analyzing system performance.

PMON presents system performance information with no kernel support. The basic mechanisms to do this task have been demonstrated. Access to kernel source, although helpful, is not necessary. The UNIX System's header files contain the information necessary to locate the kernel's counters.

System performance analysis and tuning is often more of an art than a science. Many system administrators have reported that *PMON* gives them a better feel for their systems' performance.



System Administration in a Heterogeneous Network

Bob Hofkin (hofkin@software.org)
W. Terry Hardgrave (hardgrav@software.org)
Software Productivity Consortium
1880 Campus Commons Drive North
Reston, VA 22091
(703) 648-1880

ABSTRACT

This paper deals with system administration in a large, heterogeneous network. Typical administrative functions include performance monitoring, load balancing, account maintenance, backups, and reconfiguration. Individual computers usually support these functions well, and small networks or clusters of similar machines often supported them adequately. Our goal is unified and automated support for a local network joining some 200 computers of diverse capacities and operating systems. The overall approach should scale up to networks of over 10,000 nodes.

Communication between computers has been achieved with both vendor-proprietary and open standard mechanisms. The major emphasis to date has been on application services such as remote printing and file transfer. System administration functions, such as performance monitoring, load balancing, account maintenance, backups, and reconfiguration, have taken a secondary role. Proprietary networks usually administer their interconnected computer systems, but support is often limited or nonexistent in the open networks, particularly where large or heterogeneous networks are involved. Our goal is unified and automated support for a heterogeneous network containing over 10,000 nodes.

Take as an example user account maintenance. If all users have access to all the computers, we can rely on mechanisms such as the `hosts.equiv` file or replicated password files. But there are problems if each user is authorized for a different subset of the network. A change of password or group affiliation must be propagated manually. Other changes, such as adding or removing the user, also require manual intervention.

Likewise, it is not trivial to modify important aspects of the network configuration. Addition of a node, relocation of an important resource, or change in network topology can require a separate notification for every machine in the network. Other administrative packages, such as file backup and electronic mail, also need to track these configuration changes.

Performance monitoring is another area that is neglected in typical networks. Although individual operating systems offer monitoring tools, there is not a unified view of the entire network. Tuning in the large requires the network administrator to write

site-specific tools and to correctly interpret large amounts of data.

Even the single-machine system administration tools are inadequate. At best, several independent reports can be generated from the same accounting file. Most of the tools are oriented towards accounting and operate after the fact in batch mode. We need to ask questions like "how does `cc` affect the throughput of `troff`?" and "what is causing my network congestion," but we get numbers like average working set size and the count of I/O requests. It takes considerable insight to convert those numbers to the needed information. It may be even more difficult to extract the raw information from the operating system, as such things are usually buried in `/dev/kmem` and in machine-specific units. Even if it were that easy, there remains the complication of merging disparate performance data into a coherent, network-wide picture.

It is not likely that, left to themselves, hardware vendors will correct these deficiencies. Network administration tools do not sell systems, and the system administrator is supposed to be an expert who can cope with and improve primitive tools. The administrator of a large network will not always understand the intricacies of a new system right away and may never have the time to write an adequate set of tools.

We believe that network administration requires tools capable of predictive modeling in several areas. Capacity planning is the ability to monitor system growth over time. Resource loading must be observed, both as percentage utilization and as queuing times. Capacity projections must provide an early enough warning to anticipate the purchasing cycle. The effects of software packages and collections of packages must also be modeled so the administrator can tune the

system. Tuning may improve resource usage, and so it is also a factor in capacity planning. For software developed in-house, more detailed metering of the code will let the software developers optimize their programs. The Network Integration project at the Software Productivity Consortium has as its goal the ability to manage collectively a diverse group of computers. In so doing, we intend to improve the administrative control of the individual machines by providing a general, comprehensive model of system management. Several critical technologies have been identified and are being pursued for this project. The technologies include object-oriented networking, database and query languages, rule-based analysis, and user interfaces. All of the above require both precise formal definition and intelligent implementation. Because similar tools with well-accepted machine-to-machine interfaces are needed generally, we are cooperating closely with the Network Computing Forum, an industry group with about 130 member organizations.

Network Technology

The main problem in managing heterogeneous computer systems is that of reconciling differences in command sequences and interpretations of measurements. For instance, the calculation of disk space allocated to a given user may be radically different between operating systems, and the results may be expressed in different units. Some operations will not make sense on every computer or operating system. Also, as a network grows, so does the computational burden of calculating statistics and controlling the nodes. For these reasons, the primary operational responsibility for system administration must remain on the system being administered. A higher level control program resident on a few machines would access the operating software when necessary.

One of our major goals is an open system; this goal dictated a small, well-defined interface between the operating software on each node and the higher-level control programs. The communication protocol is an essential part of this interface. We adopted remote procedure call because it is simple to use and can be provided for most any network. We further restricted the interface to client and server roles, with the operating software acting as the administrative server.

An administrative server collects local performance information in real time, and passes this information to a control program on request. The information will be converted into standard units before it is sent to the control program; various summary forms of the information may be provided. In addition, the administrative server may modify the state of its local computer, such as by starting or stopping a process, or by modifying files, in response to particular requests.

The internal details of an administrative server are not of special interest. Very often they require

access to proprietary information of the vendor, but otherwise the code is straightforward: a generic flow chart is shown in Figure 1. The Consortium is building a few administrative servers with limited functionality. A working group within the Network Computing Forum is defining the full command interface for these servers. At this time a simple query language based on HEMS [1] is being considered. Our initial network implementation is TCP/IP using the Berkeley socket libraries; we plan to convert to Apollo's Network Computing Service [2] as it becomes available.

ADMINISTRATIVE SERVER FLOW

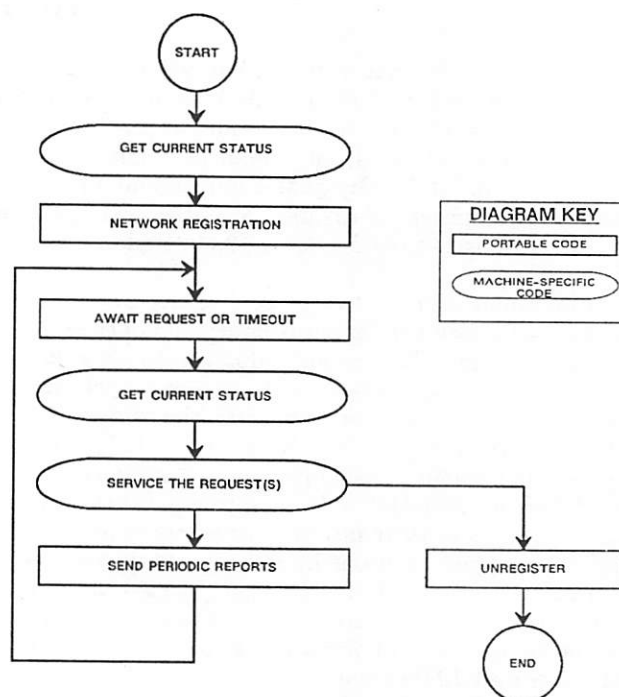


Figure 1

Database/Dynabase

Another fundamental problem of system administration is the irregular, disorganized, and ever-growing collection of information and objects to be managed. It appears that much of system administration can be translated into a data management problem that can be solved by designing an appropriate schema. We are using an entity, attribute, relationship model that is independent of particular database management systems. The schema represents objects that are of interest to a system administrator, such as nodes, routes, devices, processes, resource allocations, users, accounts, and software packages.

The schema design forces us to take a formal approach to the system administration functions, an approach that has been notably lacking in vendor-provided software and many standards efforts. The formal model will be translated into reality by the administrative servers, but the higher level control

programs can operate on a more general basis. A given vendor or site may choose to particularize the schema to better represent their view of the computer resource; database technology simplifies such an enterprise. A "core" of widely recognized entities, attributes, and relationships are defined in the schema [3], but elements can be added, removed, or modified as necessary.

A general-purpose query language such as QUEL or SQL (with some extensions) would be sufficient for our purposes. It would also be convenient to represent dimensions and units (such as "time" and "VAX 8550 CPU seconds"). Some forms of unit conversion might be provided, presumably stored as a database table. A conversion such as "VAX 8550 CPU seconds" to "Sun 3/60 CPU seconds" would be valuable in load-balancing decisions. Dimensional conversions such as "disk storage" to "dollars" can unify other aspects of a computer center's operation.

Although a database schema solves the organizational problems of system administration, current database management technology does not address the implementation issues. Current commercial databases require that data elements either be stored or be calculated from stored elements. Many elements in our schema must be calculated dynamically from primitives that are outside the schema. Some would

We propose an extension to the classical database schema so that dynamic data elements may be described. This concept is called a "Dynabase," to emphasize both its dynamic nature and its similarity to ordinary database management systems. In effect, the schema language is augmented with an additional type of computed element that specifies the appropriate request for an administrative server to execute. It may also be convenient to permit shell commands in these dynamic elements, although a shell command would not necessarily be portable. Static data is stored, and perhaps indexed, in Dynabase just as it is in any other database management system. Dynamic data is collected by the administrative servers, converted to the form appropriate to the Dynabase schema, and reported to Dynabase on demand (see Figure 2). The network administrator may view dynamic data as if it were actually stored.

We expect to use a similar mechanism to convert a Dynabase update request into changes in the running environment. There should be substantial benefit to system administrators in having a unified administrative language, and in the ability to submit a command one time and having it propagate throughout the network. The rules for updating the dynamic data elements must include additional provision for roll back or roll forward in the event of a fault. Fault tolerance and recovery is especially important in a large network environment where it is more likely than not that some important piece of equipment is down at any given moment. The current practice depends on the memory of the system administrator for this crucial step. Database technology can make the system administrator more productive and more accurate.

Rule-based Analysis

Network administration is an ideal application for expert system technology. There is a large amount of data that must be assimilated and acted on quickly. General situations tend to repeat but the particular details will usually be different. Over time, there will be an experience base developed, and there is ample opportunity for experimentation.

Initially, our inference rules are being set up to establish communication between the Dynabase and the expert system, and to develop some historical information. Typical tasks in this phase are to detect increasing and decreasing activity on an individual workstation, and to attribute the change to particular programs. Follow-on phases will add rules to detect component failures, to locate stress points in the system, and to find resources available to execute user-submitted programs.

The expert system may itself depend on database technology to store and retrieve its inference rules. Certainly the knowledge base is contained in Dynabase. The expert can interpret current performance data in the context of statistical and historical information on specific programs, nodes, users, and I/O

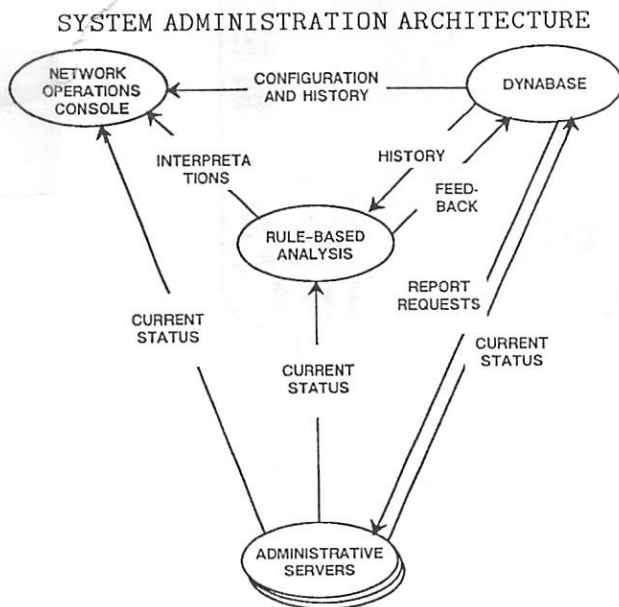


Figure 2

never be stored in the sense that a DBMS does it. We would probably expect to store the hardware configuration of a node, but information about, for example, the currently executing processes comes from system tables (as interpreted by an administrative server) at run time. Another example is the set of connected nodes. That would always be calculated, although the expected result would be stored for comparison.

devices. We expect to analyze trends such as daily demand cycles and aggregate usage changes over time. It may also be feasible to store the results of particular decisions, and so permit a form of learning for the expert system.

Conclusions from the rule base are displayed for the network operators to review. The object-oriented organization of the administrative tools permits the expert system to communicate with the Dynabase. The expert system may communicate directly with administrative servers if a closed-loop control is desired.

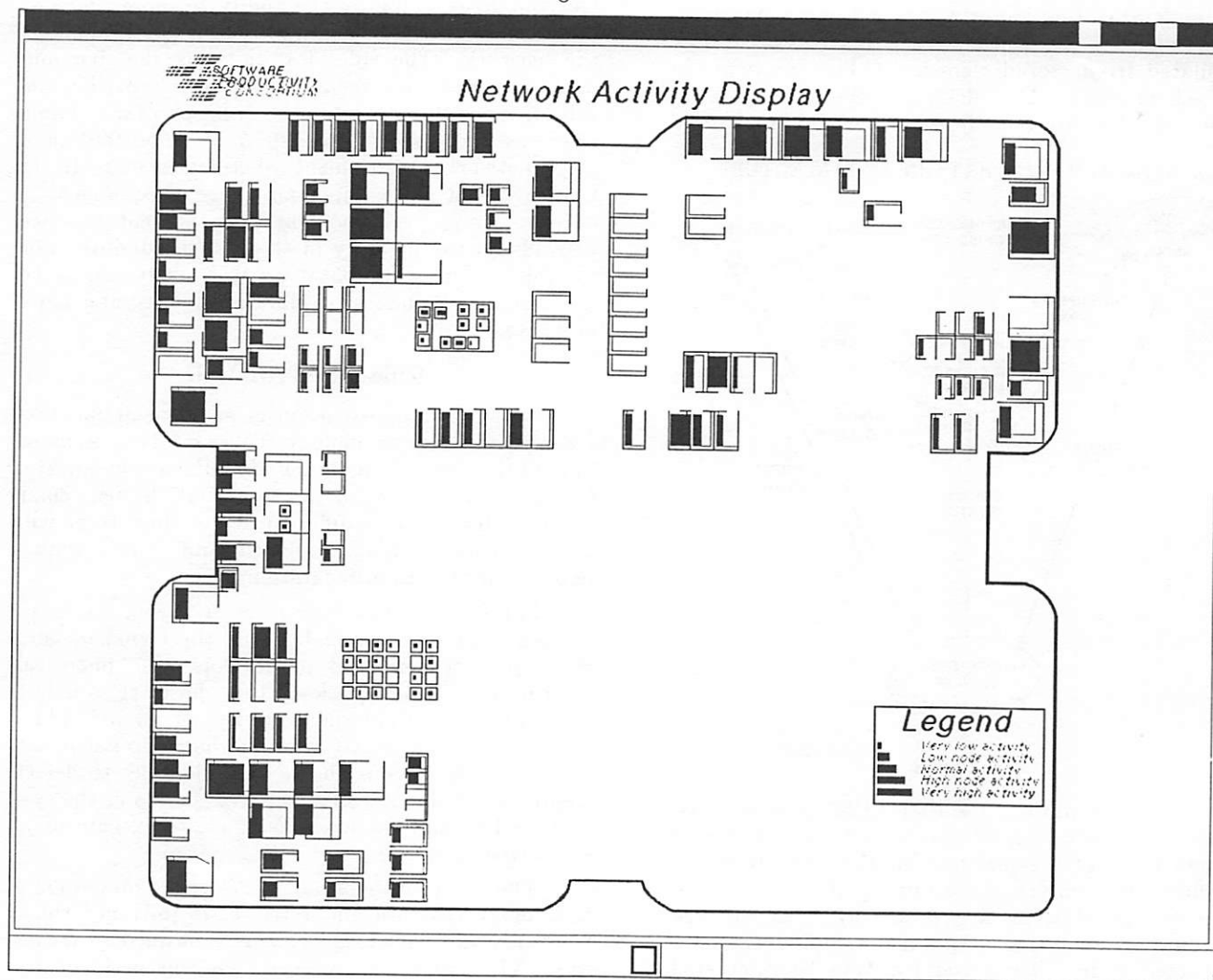
User Interface

The last major technology in the Network Integration project is the user interface, as expressed in our network operations console. We are using a graphical representation for the system components

and resources. Our main display is a physical floor plan of our office space, with the ability to show nodes, I/O devices, and cables in various colors to represent the load level. Figure 3 shows a sample of this screen (converted to black and white). Other maps under development include node and network adjacency graphs, which can be used to show traffic levels and also link failures. Additional presentation formats are planned, including scatter graphs and histograms. These graphs will be used for reports like most active programs and communication traffic source and destination. We are investigating the use of animation for the display.

Our emphasis has been on rapid communication of status to the operator, using color, size, and shape cues. We have provisions for operator control of ranges and sensitivity to various factors, but these are

Figure 3



still in a primitive state. While we will improve the user interface where it can be done easily, the mission is to provide the underpinnings for effective network administration.

Discussion

The Dynabase architecture of system administration has several attractive features. Most immediate of these is the unification of the administrative function in a consistent structure. There are several important second-order effects, as well.

Dynabase provides a common interface for an assortment of tools and reporting functions. There are many possible implementations of Dynabase, ranging from a specialized statistics file to a modified commercial DBMS package, but the interface is a schema and query language. As a result, the system administration tools can be easily uncoupled from particular computer systems. The vendors are freed from the decision to write and support comprehensive tool sets or to risk having customers locked out of important performance data. Instead, the customer can build tools locally or rely on a third-party supplier. A good commercial tool may be ported to various host computers, and additionally can support many different hosts from a single location. As installations become increasingly heterogeneous this flexibility becomes a necessity.

Another benefit for the customer is the potential for a competitive market in system administration tools. Given a common database, any number and variety of tools should be able to coexist and share responsibility for the network administration. Installation of a new tool, removal of an old one, or reconfiguration of administrative domains should not be difficult.

The computer vendors would probably have to supply an administrative server for their machine. Fortunately, it is a straightforward task to build these servers, and the maintenance burden should be low. Prototype code will be available from the Consortium and from other sources. The major hurdle will be adoption of a common server interface. At risk is the inability of a particular operating system to provide information in a form acceptable for Dynabase, or the unwillingness of a vendor to undertake the effort. Our hope is that the Network Computing Forum will establish a de facto interface which will be adopted by enough vendors to become strategically important.

References

- [1] G. Trewitt and C. Partridge, HEMS Monitoring and Control Language, Network Working Group RFC 1023, October 1987.
- [2] Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, Geoffrey L. Wyant, "The Networking Computing Architecture and System: An Environment for Developing Distributed Applications," *Proceedings of Summer 1987*

Usenix Conference, pp. 385-398.

- [3] Dynabase Schema, working paper, NCF Working Group on Formal Models and Methods for Network Administration (WG5). Draft available from the authors.



A Faster UNIX Dump Program

Jeff Polk
Rob Kolstad
CONVEX Computer Corporation
701 N. Plano Road
Richardson, TX 75081
214-952-0351
sun!convex!{polk,kolstad}

ABSTRACT

When the UNIX dump utility was written, the major concern of the authors was not speed but utility. Even the improved "token ring" dump of 4.3 BSD is unable to meet the performance requirements of today's larger filesystems and faster tape drives. CONVEX found itself with three operators - one of whom spent six hours a day mounting tapes for dump programs.

The major objective of our project was to find a way to utilize the larger memories and greater processing power of modern machines to speed the dump process. We achieve the theoretical maximum throughput of a single disk on large-block filesystems; smaller-blocked filesystems still spin 125 inch-per-second tape drives at maximum speed.

The new dump utility is completely compatible with the original in terms of options and output format while using newer UNIX features such as shared memory and asynchronous I/O. This version of the utility uses two processes, one reading the data from the filesystem into a shared memory segment while the other writes the data to tape from that memory. Both processes use asynchronous I/O; the reading process does its own raw disk scheduling to achieve maximum throughput.

A dump tape compare program and many pathological filesystem examples augmented the obvious dump/restore testing cycle.

This paper presents complete algorithms and performance results.

Introduction

In recent years, disk technology has come a long way, bringing larger and larger capacity drives to the market at progressively lower costs. These phenomena have led to a great increase in the average storage capacities across the range of computing systems. Today, it is not uncommon for any system larger than a personal computer to have many gigabytes of storage. While there are many advantages to having these large amounts of data on-line, there are some hidden costs. One of the most obvious disadvantages is the time it takes to back up all that data.

When the original UNIX dump utility was written, the authors had utility in mind, not speed. In the 4.3BSD release, some improvements were added to combat the rising time requirement for backups, but these improvements still fell short of driving 6250 BPI tape drives at their rated speeds. What was really needed was a dump utility that would make the most of today's faster disks and high speed-high density tape drives, while remaining completely compatible in format with the older versions of dump. Such a utility could exploit the larger memories and other

improvements of today's machines.

Our Approach

The solution we envisioned was to use two processes. One process would use asynchronous reads from the raw disk devices, while the other process would execute asynchronous writes to the tape. The two processes would communicate through a shared memory segment to eliminate the requirement of moving data in memory.

Rotational Latency Costs

Since throughput from the disk seemed to be the slowest link in the process, we set out to improve the methods of retrieving the data from the disks. The rotational latency seemed to be the largest consumer of I/O time. It takes approximately 16 milliseconds for a 3600rpm disk to rotate through one full revolution. This meant that the average latency time to start a disk read, even if the heads were already positioned on the proper cylinder, was 8ms. Since the read size was often just a filesystem fragment, taking anywhere from 0.3ms (512 bytes) to 5.3ms (8

kilobytes) to read, this latency was in the best case doubling the time it took to execute a read and in the worst case requiring more than an order of magnitude longer than the actual read. It seemed then that the best way to eliminate most of this latency was to issue longer reads.

A First Attempt

We stumbled ahead after deciding that the minimum length of our reads would be a the filesystem blocksize, typically 8 fragments. We first read in all of the inodes, since they are in large contiguous blocks and are fast and easy to read. Once the inodes were in memory, we would build a map of all the blocks that would be needed for the entire dump. Once the filesystem blocks were mapped, we would start reading the data in the order we would need it. But: if blocks that would be needed in the future could be read by simply extending the current read, we would do so. In fact, we reasoned that we would even be willing to read a small number of blocks that we knew we would never need in order to save system overhead by issuing a single giant read. Our first implementation was completely based on this idea and was able to achieve close to the theoretical throughput from the disk. Unfortunately, we found that this method brought about such large reads, especially on filesystems near their capacity, that the reading process would spend long periods of time reading data that would be needed in the future, while the writing process would stand idle waiting for data that it needed to continue. These idle times could approach five minutes! Once the reading process got well into the dump, the writing process would keep the tape drive running at capacity. Unfortunately, due to the waiting time at the beginning of the dump, the tape drive would continue to write long after the reading process had finished. In the end, the reading process had obtained nearly theoretical performance from the disks, but because of the initial waiting time, the total dump throughput was far from optimal and some different approach was needed to bring about more concurrency of disk reads and tape writes.

Planning Ahead

Having determined that huge reads weren't the optimal way to go (which would have been obvious from simple pencil-and-paper calculations, unfortunately), we set out to improve performance on smaller reads. We maintained our opinion that the smallest read we should issue should be one filesystem blocksize, but we also decided that we should limit the lengths of the reads to a small number of blocks and that we should not read any blocks that we didn't need. In order to obtain the throughput we desired, we reasoned that we would have to be more clever about the way we issued the smaller reads. Eventually, we decided that the reading process should form a list of blocks it needed in the near future, sort them into the *optimal order* for disk throughput, and issue the reads in that order. We

determined that the *optimal order* for the reads including sorting first by cylinder to minimize head movement/settling time and then into an order that minimized rotational latency.

To minimize the rotational latency, we experimentally determined the time it takes to issue a read request and have it propagate through to the disk controller. We would then calculate the optimal next block to read within the cylinder by adding the determined latency (in sectors) to the number of the last physical sector of the current read. The result would be the optimal physical sector on which to start the next read. We would then search the list of needed blocks for the closest sector within this cylinder that followed the optimal one, and issue that read request next. When all the requests in one cylinder were completed, we would repeat the process for the next cylinder. In implementing this scheme, we determined that even though we were grouping reads into a readv system call, the operating system was issuing a separate read to the disk controller for each element of the readv iovec. This caused our planning scheme to "blow a rev" on many of the requests and led to a greater than average rotational latency. To remedy this problem, we implemented a shared-memory allocation scheme which had the effect of a readv and issued individual (large) reads which took all factors into account.

Another problem we encountered in our machine, the CONVEX C-1, was with the number of raw disk buffer headers in the kernel. CONVEX's version of UNIX, based on 4.2BSD, allocated only one raw buffer header per filesystem. This caused asynchronous read requests to be funneled one at a time to the outboard I/O processors of the C-1 rather than queuing up multiple requests in the IOP(s). By increasing the number of raw buffer headers, we were able to halve the time required to propagate requests to the disk controller; this gained a large increase in throughput. In its finished form, this scheme provided almost the high throughput of the initial giant read scheme, without the initial detrimental waiting periods. Having accomplished our goal of achieving nearly theoretical throughput from the disks, we began looking at secondary concerns such as memory requirements.

Fragments vs. Blocks

After further examination, we determined that our assumption concerning the minimum size of the read requests was wrong. We concluded that with the clever plan-ahead scheme, we no longer needed to read any unneeded data whatsoever (even frags), and we resolved to switch the entire scheme to work with fragments instead of blocks. After solving the problems generated by fragmenting disk blocks in memory, we arrived at the final version of the over-all algorithm.

The final algorithm

The final algorithm allowed for a reading process to collect a number of needed fragments, plan the reads in the optimal order, and issue the reads asynchronously so that as soon as one physical read was complete, there was no latency in beginning the next read request. The writing process would collect tape-block sized chunks of data from the shared memory segment and issue asynchronous writes to the tape so that the next block could be collected while the write was taking place. This scheme is able to achieve close to the theoretical maximum throughput from the disks with large blocksize filesystems, and smaller block sized filesystems drive 125 inch per second 6250bpi tape drives at maximum rated speed.

Striped filesystems

The only other improvement we sought was a more clever handling of striped filesystems. CONVEX's version of UNIX allows for "striped" devices: multiple physical disks treated as an interleaved single filesystem. The CONVEX kernel handles read and write requests for these devices specially in that sequential blocks are read and written to different physical devices for increased throughput through overlap. Since our version of dump counted on knowing the physical layout of the filesystem on the disk (admittedly not the kind of knowledge a user program should have!), this special device/block mapping made the plan ahead of our dump worse than useless. We resolved, therefore, to include this special mapping in our dump program itself. We would open the raw devices themselves and map the stripe device block numbers to physical device sector numbers ourselves. This brought the throughput from stripe devices up to the level obtained from standard filesystems. Further enhancement would be possible for striped devices since requests could be issued simultaneously to different physical devices, but we have not attempted to implement such a scheme since disk throughput is already so high.

Testing

The obvious first order testing was to verify that we could restore tapes made with our version of dump. Once an "average" filesystem could be dumped and restored correctly, we began to build pathological filesystems. Besides the obvious requirement that the standard restore utility be able to read the tapes, we developed a dump compare utility that would point out significant differences between tapes created with our version of dump and those created with the standard version (and ignore such small differences as the date the tape was made). Once all test cases were passed, we began full scale use of the program internally at CONVEX, to determine if there were any special conditions we had missed, and to test it under different machine loads and various real-world filesystems.

Additionally, we created some truly pathological filesystems which included files with holes, files without holes, large files, small files, empty files, large directories, really large directories, and damn large directories. These tests exposed various problems in the dump program's interpretation of the inodes.

CAVEAT

NOTE THAT THIS dump DIFFERS IN ONE VERY IMPORTANT ASPECT FROM THE ORIGINAL. This version does NOT re-open every file just before it is dumped. If you dump an active filesystem using this dump, you run a slightly greater risk of dumping blocks which no longer resemble the file to which they used to belong. Guarantees of consistency are valid only for quiescent filesystems.

Performance

The dump program runs fast. It is just amazing to see dump tapes spin at the maximum speed of the tape drive from beginning to end without stop. Using a 6250 BPI tape drive rated at 125 inches/second, dump can run the drive flat out (just over 700Kbytes/second) whenever the filesystem blocksize is 16Kbytes or greater. Filesystems with 8Kbytes blocksize run at 60-70% of the drive's speed (400K-550Kbytes/second). Filesystems with 4Kbytes blocksize are relatively slow. They take twice as long to run as 16Kbyte filesystems. CPU utilization runs as high as 15% of a CONVEX C-1XP.

Tuning

The optimal values of some of the parameters, such as the time required for read requests to propagate to the disk controller, were easy to determine experimentally; however, there were a few parameters, such as the size limit for individual reads, which had to be determined through use on actual filesystems.

The basic rationalizations for optimizing run time go like this:

1. Reading the inodes and indirect blocks is a fixed cost which can not be avoided. Since these reads consistently report over 900Kbytes/second (almost 90% of maximum disk efficiency), it is believed they can not be improved.
2. Since the tape drives have a maximum speed which typically limits completion time, it is important both to keep the drive spinning at all times and also to start it spinning as soon as possible.
3. Individual timings of disk reading or tape writing are not as useful as may seem: the real statistic is the total real time that dump runs.

Two parameters govern the disk-reading algorithm: the "plan ahead" (which tells how far ahead to look for disk scheduling) and "maxcontig" (which tells how many blocks to coalesce if possible). It is believed that large values of the planahead parameter

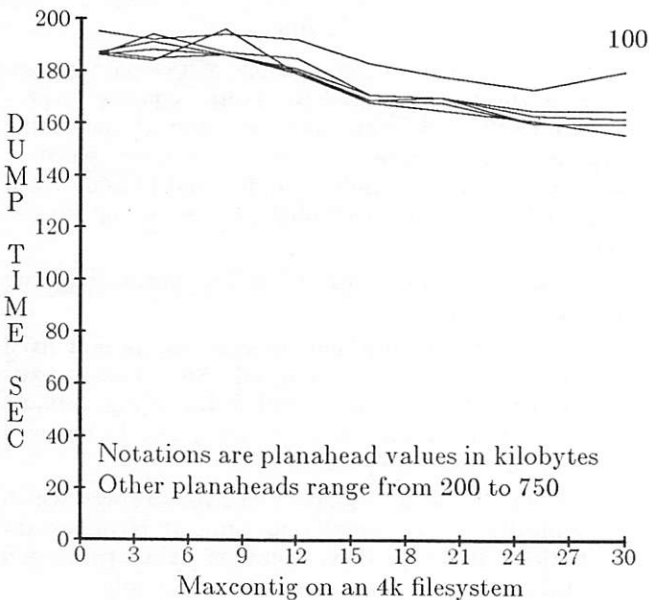
can affect the latency for tape startup. The graphs below will show that the maxcontig parameter helps smaller filesystems by avoiding blown revs but doesn't have much effect on larger filesystems.

For the tests whose results are shown below, a filesystem which consisted of the first 60 MB that tar copied from a standard /usr filesystem was dumped. The parameters maxcontig and plan ahead were varied across filesystems of differing block sizes. The 125 IPS tape drive wrote tapes at 6250 BPI. The minimum time for a dump like this is around 80 seconds.

When a blocksize of 16K (and frag size of 4K) was used, 80-83 seconds were required for 36 different combinations of the parameters. The graph is quite boring. Similarly, the 32K filesystem yielded figures within one second of 92 seconds (a bit slower since the same data occupied a bit more space on the disks). Identical results came from the 64K filesystem.

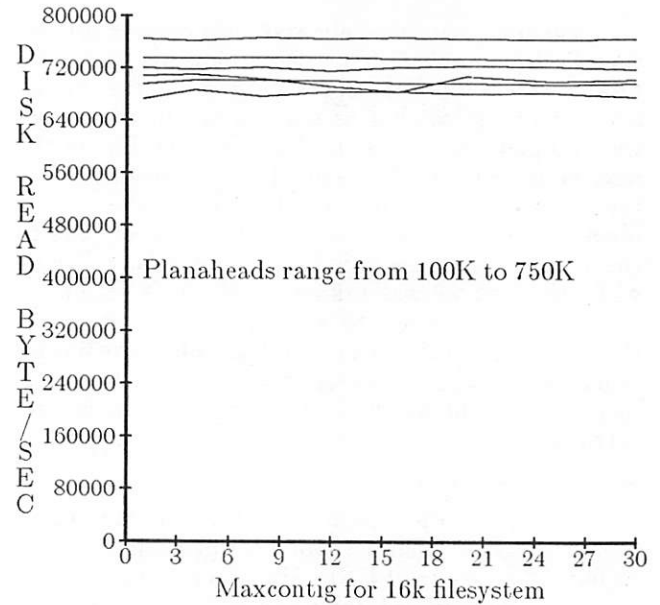
However, when the disk can not keep well ahead of the tape, the situation changes. Here is a graph showing dump time (which we would like to minimize) against the maxcontig parameter. The various curves are differing planahead values. This filesystem features an 8Kbyte blocksize:

Whenever the planahead exceeds 500-600Kbytes and the maxcontig exceeds 12-15, then optimal performance is had. Here is the graph of a 4K filesystem:



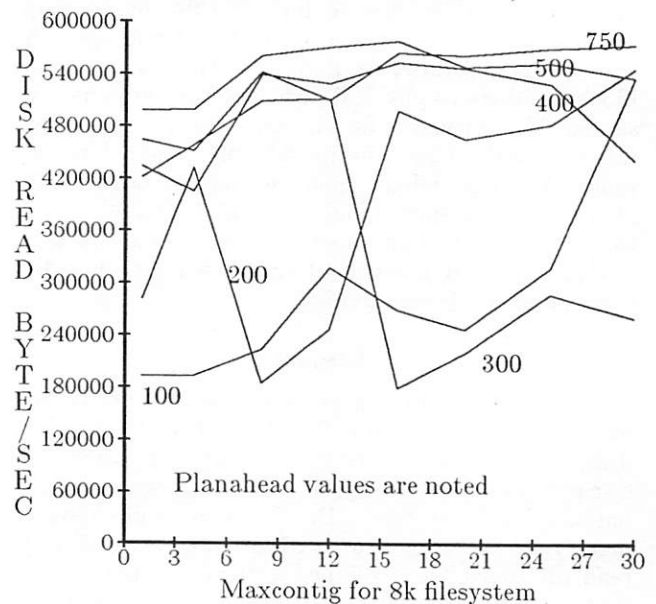
Note that maxcontig and planahead really help this situation.

Unfortunately, the system throughput graphs do not tell how well the disk really is doing (you'd need a really fast tape drive to tell). The graphs below illustrate just how the parameters affect the block reading phase:

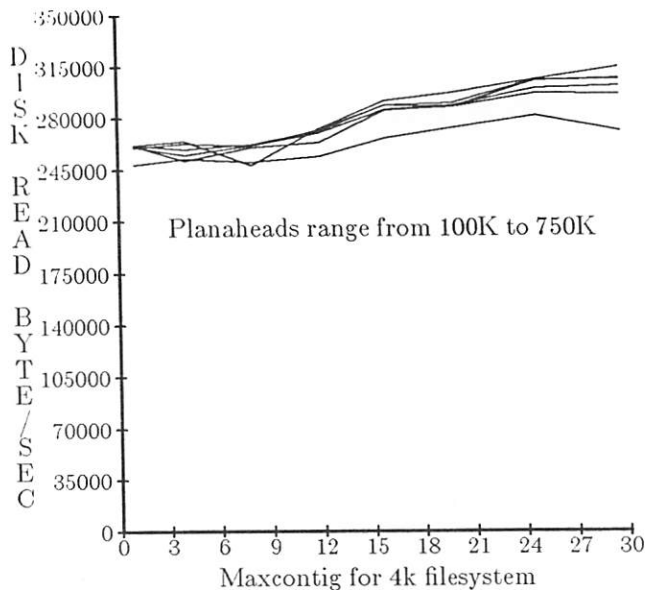


Increasing the planahead yields a 15% increase in input speed (regardless of the maxcontig value). These values were derived from statistics where the disk was required to wait on the tape if the disk went too quickly.

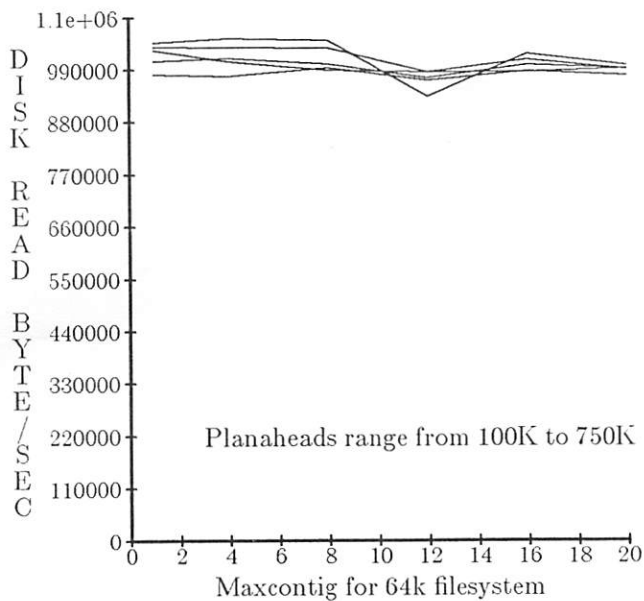
The 8K filesystem performs differently. The larger planahead values combine with the larger maxcontig values to yield ever better performance (though always 25-40% less than the 16K filesystem).



For 4K filesystems, curiously enough, maxcontig is the parameter of great influence. It is not understood currently why the above graph has such wide diversity compared to this one:



Finally, to illustrate really high speed filesystems, the program dumped the same files set up on a 64K filesystem to /dev/null. This allows characterization of only the disk-reading subsystem. The results show that larger disk blocks make reading relatively immune to parameter changes (only a few percent change). The reads proceeded at about a megabyte per second:



Conclusion

It works. One can traverse UNIX filesystems fast enough to remove much of the burden of dumps. Our machine room now completes its dumps in one third the time previously required. We do not normally unmount the filesystems but should do so for archival dumps.

Futures

We are continuing work in the area of easing the dump problem. Our current efforts focus on media which can store vastly more data than a 9-track mag tape. It is hoped we can construct a dump system which runs relatively unattended.

Once that goal is achieved, we will investigate larger stores which can be used to archive both dumps and other user files which no longer must be "close" to the processor - but which are not to be discarded, either. An automated retrieval process would complete the project.

Acknowledgments

Thanks to Jim Thompson for aiding in the initial experimental determination of rotational and propagational latencies and also for aiding in proofreading and editing this paper.

Availability

The fast dump program is not currently available for distribution outside CONVEX Computer Corporation.



Automatic Unix Backup in a Mass-Storage Environment

Edward R. Arnold
303-497-1253
era@scdsw1.ucar.edu, ...!hao!scdpyr!era
National Center for Atmospheric Research
Boulder, CO 80307-3000

Marc E. Nelson
303-497-1262
marc@scdsw1.ucar.edu, ...!hao!scdpyr!marc
National Center for Atmospheric Research
Boulder, CO 80307-3000

ABSTRACT

At the National Center for Atmospheric Research (NCAR), full and incremental backup of most Unix hosts is an automatic process. The basic elements that make this possible are a mass-storage system designed to store vast quantities of data in a heterogeneous environment, and a high-speed (i.e. greater than ethernet speed) network. This paper describes the background and features of these elements, and how standard Unix tools have been combined with them to free Unix system administrators from the burden of traditional backup methods.

The Problem

At NCAR, the direct staff in various operating divisions utilizes more than a couple dozen Unix machines as part of the array of tools they need to support a staff of scientists, and external users, in carrying out the basic mission of research in the atmospheric sciences. These machines are used for everything from production of scientific papers, to communication with universities in other states via wide-area nets such as the University Satellite Network (see Figure 1).

As with many organizations of our size and larger, the demand for computing resources has outstripped the financial ability of the organization to provide "sufficient" staff to do system administration, in the traditional way. This is particularly true with regard to guaranteeing the data integrity of machines which are scattered over four floors of NCAR's Mesa Laboratory in Boulder; traditional 1/4" and 1/2" tape drives, even over ethernet, are simply not an option.

Fortunately, the elements of the solution to NCAR's problem in this area resulted from the needs of the scientists who use NCAR's facilities, particularly the Cray-1 and Cray-XMP computers. The key elements are: (1) a repository for vast amounts of data, the NCAR Mass Storage System (MSS), and (2) a relatively high-speed network, NCAR MASnet, which can move data around faster than ethernet. The tools available on most Unix systems have

provided the glue which have made it possible to provide true "set-it-and-forget-it" backups for most Unix machines at NCAR's central site.

The NCAR Mass-Storage System (MSS)

History

The enormous quantities of data needed to support atmospheric science databases, and produced during the runs of atmospheric science models, dictated the need for NCAR's first mass-storage system in 1977. At that time, an Ampex Terabit Memory (TBM) system, based on 2-inch video tape, was installed.¹ The TBM system ran for about 8 years, accumulating some 14 terabits by the time of its retirement. Inflexible access to the TBM (it was driven by a pair of Cray-1s), and lack of channel capacity, forced planning to begin for a more versatile system by 1983.

Planning for the Current MSS

Planning for the current MSS began in 1983,² and it was found that there were no systems at that time which met all of NCAR's needs. When actual design work was begun in 1984, the preliminary specifications were oriented around write-once optical disk, although as few assumptions as possible were made about the hardware configuration. The design was heavily driven by similar systems, particularly the Reference Model³ and Los Alamos CFS⁴, by what computing hardware was available to which it was likely that an automated library could be retro-

fitted⁵, by expandability of the host computer on which the MSS was to reside, and by the need to procure hardware which would be supportable over a long period of time.

The Selected Hardware & Software

The hardware and software that were selected to create the MSS, partly shown in Figure 4, includes:

- An IBM 4381 CPU with 16 megabytes memory, termed the MSCP (Mass Storage Control Processor).
- Twelve IBM 3480 cartridge tape drives, each dual-ported to two controllers; these are currently operated manually.
- Twenty-four IBM 3480E disk drives, whose primary function is to provide a buffering "disk farm" of approximately 100 gigabytes.
- A Network Systems Corporation A222 HYPERchannel adapter, for access to the MSCP by systems other than NCAR's Crays. (The Crays are very tightly coupled to the MSCP via multiple HYPERchannel adapters and IBM channels, reflecting the needs of atmospheric science; other access to the MSCP is via either one or two 50 Megabit/sec HYPERchannel trunks, depending on remote system location.)
- Various miscellaneous pieces of hardware, such as 1/2" tape drives that allow the importing of data to the MSS, when electronic transmission is not feasible.
- An MVS-XA operating system.
- About 70K lines of PL/1 & BAL to implement the basic MSS design, about 35K lines of FORTRAN to implement fast transfers to the Crays (not of direct interest here), and about 15K lines of FORTRAN to implement the interface to all other machines via the NCAR Local Network (also known as MASnet).

The User View

Many of the characteristics which the end-user of this MSS sees, are characteristics which were first popularized by Unix. The primary differences between the NCAR MSS and a Unix filesystem, are extensions that are dictated by the NCAR operating environment, the fact that some files are automatically migrated off-line when needed, and the quantity of data stored.

The most noticeable characteristic of the MSS filesystem, is that it is tree-structured; each file entered in the Master File Directory (MFD) contains a pointer to its parent, and is on a chain of pointers for items at its own directory level. At the root, each subtree begins with the username of an MSS user, and is dependent on the user for organization below that point. For instance,

" /BOZO/ARECIBO/VEL1985"

is an actual user file which contains data from the radio telescope at Arecibo, for the year 1985.

"ARECIBO" is a directory which is used to isolate Arecibo data, from other projects. One difference between the MSS tree, and a Unix tree, is that there are never empty directories, because the MSS automatically removes them.

Beyond the tree-structured filesystem, many of the characteristics all Unix users are familiar with, somewhat extended, can be seen by looking into the MSS' Master File Directory, or MFD. The MFD contains many items analogous to those in a Unix inode:

- The familiar file types of regular file (here, termed a bitfile³), and directory, are present. However, because the MSS has to deal with the concepts of putting users files on particular media, grouping media, dealing with non-random access media (tape cartridge, optical disk), and commanding a human operator or robot to mount or dismount media, it has been necessary to add types of "medium" and "medium header"
 - Times of creation, last read, last write, and last reference. The reference time, which is somewhat analogous to Unix inode "access time", is a composite time resulting from reads, writes, or touches, and is the base to which retention period is added to find the time of purge.
 - In addition to the above, due to the need for statistics that will aid in managing the MSS' very large MFD, each file includes a time of last compression (discussed below), three previous times of last reference, and counters of the number of times read and written.
 - Every file is flagged with an owner number, like Unix "uid". This is slightly extended, however: every file is also tagged with the user name, to simplify finding the owner manually.
 - A flags field, analogous to Unix mode bits. This provides the concepts of owner and group read and write permissions, the precise use for which are still under design as of this writing. The fact that these aren't currently used, has to do with the per-file passwords.
 - Pointers to the location of the data, on either disk storage (primary) or removable medium (secondary). At any given moment, data can live on either or both of primary or secondary storage, and is made to reside on secondary storage, at some point, by automatic migration software.
- Some of the most noticeable differences between the MSS filesystem, and a Unix filesystem, are:
- There is a retention period, since the vast quantity of data dictates that something needs to be done about getting rid of that which nobody cares about.
 - There is currently a data format (unlike the Unix concept of a byte stream), which is really a historical accident. This resulted from the fact that NCAR's environment uses the format of a dataset under Cray's COS operating system as the basic medium of interchange (external data

representation). Since the shortcomings of this approach were recognized at design time, however, the MSS' data format field has already been set-up to accommodate an "unknown" type. Within the next year, the MSS will allow a more Unix-like approach in that it will stop knowing the format of some files (more in keeping with the definition of a bitfile³), thus eliminating all format translations on such files.

- Each MFD entry retains a record of the name of the computer on which it originated. This makes possible, for instance, automatic notification of users whose files are about to be scrubbed, via electronic mail.
- Each file can have read and write passwords. This facilitates selective sharing of data between users, esp. when a user wants to be able to give someone else access to a single file, without having to create a group. Many users do not utilize these, but the very public nature of such a large repository makes them useful.

Maintenance Features of the File System

Some of the most noticeable differences between the NCAR MSS filesystem (some of which were discussed above) and a regular Unix filesystem, are those features which deal with the much greater maintenance requirements of the MSS filesystem. The most important of these are:

- **Automatic Purge:** Due to the vast quantities of data, a way of automatically removing files anywhere in the system was provided to deal with the problem of data that has a very limited lifetime. Every file is assigned a user-specifiable retention period which is always relative to the last file read/write/reference/touch. If the file is not referenced within the number of days specified by the retention period, it gets scrubbed.
- **Compression:** Unlike a standard Unix system, the NCAR MSS has to deal with the problems caused by non-random access media and, given its design history, write-once media. Although end-users are normally not aware of it, data on tape cartridges is constantly being moved around to fill "holes" left by deletion of files. The system itself is responsible for many deletions, because it is the policy to write a new copy of an existing file in a new location, and delete the old copy from wherever it was.
- **Data Integrity:** Due to the trust placed in NCAR's MSS by a very large community, including system administrators who use it as a repository for backup of their systems, the MSCP runs a very low priority data integrity checker program in background, which reads files essentially at random to determine if corruption is evident. This feature has been very useful in finding software and hardware problems before they can result in significant damage.
- **Data Migration:** At certain intervals, or when the approximately 100 gigabytes of disk farm

reaches an occupancy threshold, data migration to tape cartridge automatically begins under control of a low-priority background program. Users on front-end computers who need to make sure their data is on-line, can issue commands to the MSS via MASnet to force it online. In practice, it has been found that backup archives sent from Unix machines tend to stay online for at least several days, often over a week, and the delay involved in cartridge mount to bring material on-line has never caused any significant delay or inconvenience to users.

MSS Status and Future

There is no question, at this point, that the MSS has been eagerly accepted by NCAR's user community. The system was initially brought on-line, in a small configuration, in second quarter 1986. By the end of 1987, the system was storing some 45 terabits, with about 38,000 3480-type cartridges in use, and approximately 1000 cartridge mounts by operators each day.

It is likely that the MSS' future will include other types of storage devices, such as video or optical devices, and automation of cartridge mounts.

A Multi-Network Environment

As important as NCAR's MSS has become to the atmospheric research and programming staffs of the institution, it is nearly as important that our staff have a way of moving data to and from the system in a high-speed manner. NCAR's networking began about a decade ago, with HYPERchannel adapters on 50 megabit/sec trunks. Today, this system has evolved to complement a traditional ethernet LAN in some important ways.

NCAR uses two networking systems in parallel: standard 10 megabit/sec ethernet, supporting TCP/IP and DECNET protocols, and 50 megabit HYPERchannel, running NCAR's MASnet software, as illustrated in Figures 1 and 2. In general, ethernet carries "human-sized/usable" files: electronic mail, telnet sessions, and the like, whereas MASnet handles multi-megabyte files that happen almost exclusively in the "background". Typically, these multi-megabyte files consist of data to be perused on a front-end computer sometime after a lengthy simulation has been finished, such as graphics metacode files, for which human interaction is not needed while they are being created. More recently, these files have come to include large files from Unix front-end computers, whose system administrators cannot afford to back-up in traditional ways.

Some of the characteristics that make MASnet preferable to ethernet for system backup are:

- **Transmission Speed:** In most cases, MASnet offers higher transmission speed than ethernet, although (being CPU-bound), the speed realized is not startlingly higher than ethernet. When conversion to external data representation is not done,

MASnet offers a disk-to-disk rate peaking at about 200 kilobytes/sec.

- **Channel Contention:** More important than end-to-end speed, is the matter of channel contention. With a number of systems to back-up at approximately the same time, and usually while a small number of staff members are still working on various machines, tests have shown that several simultaneous, long ethernet transmissions could cause severe degradation through excessive collision. However, MASnet uses large packets, and the actual time in transmission at 50 megabits/sec is short enough, that contention is never an issue.
- **Bridging:** NCAR's ethernet environment has gradually been expanded to include long-distance satellite and land links which extend as far as both coasts (see Figures 1 and 2). This, and increasing ethernet usage within NCAR's divisions, has led to the use of intelligent bridges between sub-nets to maintain contention on any segment at a reasonable level. Thus, were the MSS connected to ethernet, attempts to copy large files across one or more bridges could create problems with bridge loading.
- **Backgrounding:** Activities such as very large model simulations and system backup are inherently background activities. The design philosophy for MASnet is inherently a good match for these types of activities.

This is not to say, however, that slower-speed LANs such as ethernet could **not** be used for large-packet background transmission; such activity demands a network on which interactive response is not an issue. In other words, NCAR's needs could probably be met by installing separate specific-purpose ethernet network(s), were MASnet not already in place.

Network Software

Historically, NCAR's network software has been oriented around the ideas of performing file transfers (akin to ftp or rcp) and job submissions (akin to a background rsh) in a very heterogeneous environment. These basic capabilities have been extended to some very diverse types of operating systems, including IBM/VM, IBM/MVS, Cray COS, VAX/VMS, RSX-11, and a couple flavors of Unix.

This networking scheme has several layers:

- 1) A network executive, whose job is to transfer data between itself, and the network executive on some other machine. The only thing the executive knows, is that the first segment of data it receives from the program which called it, is a "control segment". A limited number of fixed parameters in this control segment tell the executive certain important things, such as the remote node involved.
- 2) Programs called "surrogate" (where a request is transmitted) and "post-processor" (where a request is received), that do the actual work of

data conversion, reading/writing files in the local filesystem, etc. These programs call, or are called by, the executive.

- 3) User-level programs, which do useful work for users by setting up the control segment in some way, then calling the surrogate to do data conversion and set up actual transmission with the executive. In fact, the ability of the post-processor on any given machine to understand what has been packed into variable-length parts of the control segment by a user-level program, is what has allowed expansion of the job MASnet does, from simple file transfer, to some rather elaborate requests of very specific type, e.g., send a job stream to some other machine, execute it, and send its output back to the original machine. The most recent versions of this program, allow execution of a completely user-specifiable program on the remote system.

Basic Access Through Netxx

For Unix nodes on MASnet, almost any job that can be understood by the post-processor on the targeted remote node, can be typed in as a command that neatly packs away every parameter into the control segment header, the variable-length part of the control segment which applies to the local node, or the variable-length part of the control segment which applies to the remote node.

This very low-level (perhaps level 4 or 5 in the OSI model) request is very general. The only problem with it is that nobody who has ever used it, can remember what to type in the next time to do a job, nor do they want to!

Expanded Access Through MSS-Specific Commands

Very early in development of the MSS, it became obvious that there weren't any users who would want to submit elaborate job streams to the MSS, to find out the status of their files.

Therefore, any front-end Unix machine at NCAR which supports any significant number of users, has a small number of commands which perform a set of orthogonal operations on MSS files, and return the result to the user on his home computer, usually in real-time. On Unix machines, the set of commands which is directly analogous to Unix commands is:

msls - list names & attributes of a file tree
msmv - change name of a file on MSS (may require per-file passwords)
msrm - remove an MSS file (may require per-file passwords)
mstouch - change time of last reference of an MSS file
toms,fromms - (like rcp) move files between the local filesystem and MSS

Since the MSS contains some file attributes not found in Unix, the following commands are extensions:

mschg - change password(s) or retention period associated with an MSS file
 msonline - queue request with MSS to mount a cartridge, & bring a file online
 msxport, msimport - queue request with MSS to transfer an MSS file to/from conventional (1/2") tape, or IBM 3480 cartridge

The Backup Front-End

NCAR's MSS and networking provided the basic facilities we needed to implement automatic backup. The third component, a "front-end" program which utilized these tools, was created by glueing together common Unix tools, mostly at the shell level.

There were several reasons that a "front-end" was needed:

- Network transaction time/overhead precluded individual copying of files.
- The MSS was designed to store relatively large files. It didn't make sense to clutter the MSS' MFD with thousands of small files.
- Our environment is heterogeneous, and the MSS could hardly be expected to directly store every type of file provided on every operating system we run. For instance, the MSS cannot store the attributes of a special /dev file directly.

The requirements that the front-end had to meet were:

- 1) It had to be runnable on all Unix systems from V6 to 4.3.
- 2) Where a system was not attached to MASnet, it had to be able to gateway backups through another system which (1) supports MASnet, and (2) is attached to the same ethernet segment as the system to be backed-up.
- 3) It had to be put together quickly and be easily modifiable, so most of it had to be written as a shell script.
- 4) It had to use methods which allowed restoration while the system of interest was up, in multi-user mode.
This meant that filesystems could not be copied out "raw", and archives to be brought back had to be of a convenient size to fit into temporary space.
- 5) It had to save sufficient information about each backup to detect corruption introduced by either the MSS or network, and at least name information about the files contained in each backup had to be retained on the backed-up system.
- 6) It had to be efficient enough to complete a full backup within a few hours.
- 7) It did **not** have to deal with the issue of what to do with files that are being written at the time a backup runs. Fortunately, we don't write large database files in the middle of the night.

The general scheme devised is shown in Figure 3. A master list is constructed, and this is broken into

sublists from which archive files of reasonable size can be constructed. User-level network software is then called to transport each of these archives to the MSS, catalogued in a hierarchical arrangement shown in Figure 3. The sub-lists are retained on the backup system, in a hierarchical arrangement analogous to that on the MSS.

Unix Shortcomings

In the process of putting together this "front-end", we found that the Unix environment was not quite as versatile as needed in several areas.

The "find" command was really the only reasonable existing way to generate the master list. However, the inflexibility and granularity of time expression with find was a minor problem, as well as the lack of any way to compute time intervals at the shell level. This issue was circumvented by "overcoverage" in backup.

Both major archival programs (tar and cpio) had deficiencies, but tar was found to be the most deficient; for instance, its inability to restore directory modes other than 755 correctly, was unacceptable in our environment. For this reason and others, we find the apparent non-inclusion of cpio in the POSIX standard to be somewhat troubling.⁶

We could find no program in the Unix releases available to us, which was capable of splitting a file list into sublists, with the number of bytes in the files of each sublist the parameter of interest. Therefore, yet another variation on "split", called "szsplit" was written, and has since been found useful in maximizing the amount of data placed on tapes in ordinary tape operations.

Finally, there really was no way to reconcile the need to pipe cpio output to network transport software, for efficiency, with the need to collect the checksum from each archive file via the "sum" command at shell level. Since efficiency has not been an overriding factor in our environment, we've simply created the archive files in temporary space at the cost of elapsed time.

Backup Restoration

A big plus with a networked automatic backup system is that the system administrator can perform restorations without ever having to leave his terminal. When a restoration request is received, usually by filename, the on-line name lists can be searched, and the corresponding archive brought back, within a few minutes. Time-consuming manual searches of a conventional tape library are never necessary.

At some point in the future, it is possible that the number of requests, even with this level of automation, will grow burdensome. Because manual intervention by the system administrator isn't necessary, a totally automatic system which is capable of accepting backup requests from users and restoring, may eventually be built.

Summary

The NCAR MSS has made it possible for Unix system administrators at NCAR to provide frequent and thorough backups for their systems, which would not otherwise be possible. At the same time, the labor cost of doing system backup has been drastically cut in comparison to traditional backup methods.

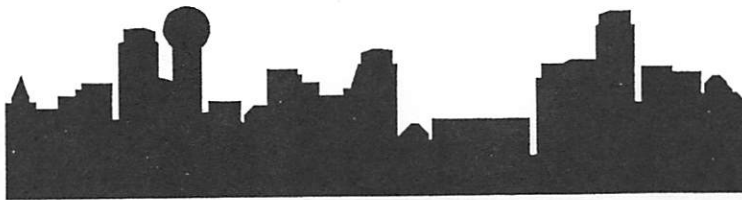
Now that the basic concepts of the MSS, especially automatic migration to a secondary medium and the cost benefits of centralization of data storage operations, have been proven, it is clear that this general approach to centralized data storage offers many benefits to other organizations, especially those of medium to large size. For these reasons, the rights to the NCAR MSS design have been transferred to Mesa Archival Systems Inc. through the UCAR Foundation, (the University Corporation for Atmospheric Research is NCAR's parent), and it is expected that the technology will eventually appear in commercial products.

Acknowledgements

Thanks are due to both Paul Rotar and Bernie O'Lear, for providing the management support that helped make the MSS a reality.

References

1. Marc Nelson, David Kitts, John Merrill, and Gene Harano, "The NCAR Mass Storage System," *Digest of Papers*, Proc. Eighth IEEE Symposium on Mass Storage Systems, May 1987, pp. 12-20.
2. Paul Rotar and Roy Jenne, "Mass Storage System - Requirements and Projections," *Proc. Third Annual Computer Users Conference*, NCAR Technical Note (NCAR-TN/219+PROC), November 1983, pp. 90-107.
3. Stephen W. Miller, "Mass Storage Reference Model: Version 2.0", IEEE Technical Committee on Mass Storage Systems and Technology, May 1987.
4. Tyce McLarty, William Collins, and Marjorie Devaney, "A Functional View of the Los Alamos Central File System," *Digest of Papers*, Proc. Sixth IEEE Symposium on Mass Storage Systems, June 1984, pp. 10-16.
5. Patric Savage, "Proposed Guidelines for an Automated Cartridge Repository," *Digest of Papers*, Proc. Seventh IEEE Symposium on Mass Storage Systems, November 1985, pp. 53-61.
6. Usenix Association, *:login:*, July/August 1987, page 13.



System Administration Daemons

Von Jones
Convex Computer Corp.
701 N. Plano Rd.
Richardson, Texas 75081
{ihnp4,sun,uiucdes}!convex!vjones
214-952-0324

ABSTRACT

This paper describes a set of system administration daemons which we use at Convex Computer Corporation. The programs described help us to reduce the time we spend on system administration, enhance user satisfaction with their computer environment, enhance user productivity with respect to their computer environment, better utilize system resources, detect security breaches, and decrease downtime.

Introduction

As a commercial system administrator, I address many problems which affect the security and reliability of the systems I administrate. My job is to provide users with a computer environment which makes them as productive as possible. The operations team which I am a part of administrates eight convex C-1's, one Vax 750, one Sun 3/160, one Sun 3/180, ten Sun workstations (3/50's and 3/75's), and three ethernet connecting these and other cpus around the company.

We provide operations service for approximately three hundred people, and address aspects of global operations, such as electronic mail, our three ethernet, and all of the terminals used in the facility, for over five hundred employees. We have a combined disk usage of over twenty gigabytes. In a hardware and software development environment such as ours, the hardware and software which constitutes each machine is in a constant state of flux. We therefore encounter system administration problems at an accelerated rate.

Among the things I must ensure on a day to day basis are security, availability of resources, employee electronic communication, getting needed information to users, coordination of resources, and general system maintenance. I am expected to be the expert on the systems I administrate and to address a great number of problems daily. To this end, I have attempted to automate as much of my job as possible. This automation frees up more of my time to deal with problems such as coordination of resources, future planning, and user questions, which, by their nature, require human intervention, and helps our systems to run more smoothly.

Among the programs which we use, I feel that one class, daemons, has been of particular importance. These programs run in the background of the

system and address many areas of system administration. I have tried to write daemons which address the most frequently occurring problems I encounter. As a general rule of thumb, if I encounter the same problem three times in a month it's often worth writing a daemon to address it. Most of the daemons which I will describe are easily written. Many of them are shell and awk scripts. Some run from crontab at an interval suited to the problem which they are trying to address. Others run on an interrupt basis, usually running idle but occasionally waking up to process available data.

The data which the programs require is taken from the system state and from a file which contains information on user accounts. The system state is affected by such factors as the state of the hardware, the state of the filesystem, the contents of system files, processes being executed, or the number of users. Information on the system state is by probing system files or by analyzing the output of existing commands. Many system files such as `/usr/adm/acct`, `/usr/adm/badlogins`, `/usr/spool/uucp/SYSLOG`, and system error logs contain a wealth of information but are often only perused when the cause of some sort of error condition is being sought out.

By writing programs to analyze and take action on the contents of these files, an administrator can achieve much better control of problems which exist on his system. Often there already exist programs which analyze existing file system data which can be used to build larger programs and which provide much more complex analysis. Efficiency of execution would, of course, be better obtained by probing having individual programs probe the system files rather than calling other programs which access them. In many instances, however, a system such as ours can be brought up quickly by utilizing small shell and awk scripts at the expense of efficiency. With the time

which these programs should free up, you can later enhance efficiency of execution as needed.

The other source of data for these programs is the *users* file. This file contains login name, the numeric user id, finger information, default shell, accounts and their associated quotas, whether or not the user is a Convex employee, the user's default password, and the machine where mail for the account is delivered to. At this time we are still small enough to keep all of this information in text format. Keywords *name*, *added by*, *uid*, *alias*, *office*, *ophone*, *hphone*, *fullname*, *group*, *shell*, *passwd*, *host*, *unlist* and *last_edit* identify the information contained in the line, and make the file easily parsable by shell and awk scripts. Blank lines separate the records which constitute the database.

Each record element can occur in any order within the record, and records can occur in any order within the *users* file. All entries have a single data element except for *alias*, *host*, *unlist*, and *last_edit* entries. *Alias* entries have 2 data elements: the home machine onto which the data is delivered, and the person to deliver to on that machine. *Host* entries describe the machine where an account exists, its quota, and the directory which constitutes the account. The last host entry for a specific system is the login directory for the account on that system. *Unlist* entries are optional. If an *unlist* entry is present then the user is not a Convex employee, and would not be listed in the on-line phone list. *Last_edit* lines identify the user who last modified the users entry, and the time of the modification. Here is the format of a *users* entry on our system:

```
name username
added_by administrator
uid universal_decimal_uid
alias home_mail_machine username
office office_location
ophone extension
unlist
hphone home_phone
fullname users_full_name
group group_name
shell default_shell
passwd initial_encrypted_password
host machine1 nblocks directory1
host machine1 nblocks directory2
...
host machine2 nblocks directory3
...
host machinen nblocks directoryn
last_edit login_name date
```

Users entries are separated by one or more new lines. The *users* file resides on one central machine. This makes the file more secure, makes file locking easier, and facilitates the coordination of user ids and other information when creating accounts. Two utilities, *nu* and *eu* allow easy editing of users. *Nu* creates new users by prompting for pertinent information and appends the new entry to *users*. It will not allow

creation of two users with common user names, and it is responsible for passing out unique user ids. *Eu* invokes your favorite editor on a single user entry, and puts the edited text back into the *users* file if it has been modified. Yet another program, *getuent*, obtains a user's entry out of the *users* file and prints it to standard output allowing easy read-only access to individual user entries.

The text format of the *users* file is perfect for use with awk and shell scripts. In fact, *getuent* is a perfect example; it is nothing more than a small awk script. A more execution efficient approach would be to keep the information in a database format. Such a database format could be more efficient in larger environments, but would add to the complexity of the programs which use the data. Since most of these programs run when the system is idle, the speed with which they can be created and maintained is often more important than their execution efficiency. Balancing these factors on your own system will depend on the constraints which you work under.

The Daemons

Monitoring Daemons

Traditionally, administrators find problems from their own use of the system or from user reports of system problems. One class of the daemons which our team of systems administrators use probes the system for potential problems and reports any problems encountered to the system administrator(s) responsible. There are many reasons why this is vastly preferable to traditional means of finding problems.

First, many problems can be encountered while they are small and virtually unnoticeable, thus allowing the administrator to solve the problem before it becomes large enough to have a serious impact on the system. This can easily avert some system disasters, such as */usr/spool* filling up, bad spots in a disk, or security violations.

Second, it allows the administrator to address the problem before s/he gets calls about it from annoyed users. This decreases the number of problems which users perceive that the system has, thus increasing their satisfaction with their work environment and making them more productive. Even if an administrator doesn't have time to address a problem immediately, s/he can at least talk about the problem intelligently with users who might report it, and give them confidence that their problem is being addressed. Further, these programs have dramatically decreased the number of interrupts which I receive when users report system problems, thus providing me with greater continuity of thought and helping me keep my sanity.

Finally, problems which administrators or users might not normally notice can be found and addressed as desired.

Resource Monitoring Daemons

Disk Monitors

Many of these system probing daemons insure that system resources are available for use. The most analyzed system resource is disk space. There is great difficulty in managing the disk space of a large number of users.

This difficulty is compounded by the dynamic software development in which I work. Through methodical attack of some of the largest factors in disk wastage, we have been able to keep our disk consumption at a reasonable level.

Two daemons collect the data with which other disk daemons access: *diskspace* runs nightly and determines actual usage on the system, and *unacc_rep* runs weekly to determine the amount of disk in each account which has been unaccessed in over some period of time. *Diskspace* runs on each system and computes the actual amount of disk (in blocks) being used by each account. The output file is organized in a one account per line fashion.

Each account's name is paired with the name of the account, one per line, and the lines are saved in a file */usr/adm/disk.today* on the machine on which they run. These files are then collected on a central machine (the same machine which the users file resides on) for processing. Block totals for each account are provided by running the standard Unix utility *du* with the "-s" option, thereby only producing totals for the directories it analyzes. *Diskspace* is easily written and maintained. *Unacc_rep* reports the number of files in each account on a machine which have been unaccessed in a given period. We use a period of ninety days. It runs by executing a *find* on each account to find files which have been unaccessed and piping the output through an awk script to find the totals. The *find* commands must fence themselves out of nfs partitions in order that meaningful data be provided.

Users are provided with a command *unaccessed* with which they can determine the actual files which are in question. Obviously this could be made far more efficient by writing a C program, but since it only runs weekly and even then in the middle of the

From daemon Fri Nov 27 08:09:03 1987
To: vjones
Subject: disk report (from disksumm program)

This mail is computer generated. It goes out once a week. Please examine the statistics given. Be on the lookout for an excessive increase in disk usage which you cannot account for, for files which have not been accessed in a long time and could be taken off onto tape, for accounts where you have exceeded your disk quota, and for accounts which are obsolete. Disk space is pretty tight and we need to keep our use of disk space to a minimum. Quota is the disk space allocated to the account, usage is the actual disk being used, -1 weeks is your usage a week ago, and unacc is the amount of disk space no one has used in any way for 90 days. A block is 1 kilobyte (1024 characters)

account (machine:directory)	quota (blocks)	usage (blocks)	-1 wks. (blocks)	-2 wks. (blocks)	-3 wks. (blocks)	unacc 90+ days (files/blocks)
john:/mnt/vjones	1000	107	107	61	61	3/27
paul:/mnt/vjones	1000	1072	543	763	1242	4/250 +QUOTA
george:/mnt/vjones	1000	154	154	97	97	7/12
ringo:/mnt/vjones	1000	200	200	154	154	1/9
richards:/mnt/vjones	10000	9543	9253	10260	8776	28/3623
watts:/mnt/vjones	500	77	77	171	171	6/7
jagger:/mnt/vjones	500	295	295	201	201	4/82

Please see if you can archive the data in the following accounts:
richards:/mnt/vjones

NOTE: There is a utility 'unaccessed' in */usr/local/bin* which will report on all of the unaccessed files in a set of directories you specify. Call it as 'unaccessed <directories>'. Note that the unaccessed command is better run at night for large directories as it causes LOTS of disk seeks.

Figure 1

night when systems are largely idle, we have not yet done so. If it were not our intent to provide unaccessed information on a per-account basis, the *quot* utility, which can efficiently produce reports of unaccessed space grouped by user for a given set of file systems, could be easily utilized to provide the same motivation to conserve disk space as our per-account system does. We choose this per-account system since it gives a better picture of the actual use of disk space on a per-project basis. For our purposes this per-project accounting is much more appropriate.

The file */usr/lib/fullfs.thresh* also provides data to drive the disk analysis programs. It is a static file which exists on each system and describes thresholds of capacity beyond which each filesystem should not go during normal operation. The entries are lines describing the filesystems to be analyzed. Each line contains the name of the partition and the percentage of usage of available space which is that filesystem's threshold. The rest of the information which drives the disk analysis programs is taken from the output of Unix commands.

Our principle disk analysis programs are *fullfs*, *disksumm*, *quota_mail* and *reaper*. *Fullfs* warns administrators when a file system is almost full, allowing them to fix potential problems before users encounter a lack of resources. It decreases the number of user and system writes to disk and therefore increases both user and system productivity. *Fullfs* merely takes the output of the Unix command *df*, along with the contents of the *fullfs.thresh* file and determines which filesystems are overly full. If a filesystem is overly full *fullfs* sends mail to administrators responsible for maintaining that system's filesystems.

An awk and a shell script accomplish this. Ideally, however, *fullfs* would do a lot more analysis, such as detecting large swings in usage and informing the administrator when a threshold is not set for a given filesystem. *Disksumm* is a tool which we use to help users to better analyze their own disk usage. By doing so it both encourages users to take a more conservative and more responsible approach to disk usage, and saves our company money which would be spent on unnecessary disk expansion. *Disksumm* mails users a weekly report on each of their accounts with quota, usage, and unaccessed file information. It highlights accounts which are over quota and accounts which are good candidates for cleanup (they have a number of blocks unaccessed in ninety or more days which is more than a given threshold -- our threshold is one megabyte). *Disksumm* is written in C and uses previously collected data from the *users* file and from the output of *diskspace* and *unacc_rep* to build a report for each user. Exceptional conditions such as long periods of time over quota, a lack of information, or an excessive amount of unaccessed disk space are included in an overview report provided to administrators.

A piece of mail from *disksumm* looks something

like figure 1. Our quota system, with its main daemon *quota_mail*, mails out over quota messages daily to over quota accounts, as well as producing a daily report of all quota problems. We prefer this mail-based approach to a quota system which complains on every write and has a hard quota beyond which a user cannot write. This is because the mail system is much less of a deterrent to user productivity. Both *disksumm* and quotas have encouraged our users to take a more responsible view of disk conservation.

Reaper attacks the problem of unneeded files by removing old editor checkpoint files, messages, *dead.letter* files, and core files which would normally clutter a system. It removes *ruho* files referring to systems which have been out of service for extended periods of time, and it clears the */tmp* and */usr/tmp* partitions of old, unused information. *Reaper* is run nightly by *cron*. *Find* commands perform the work of this script. The *find* commands must carefully keep themselves out of all *nfs* filesystems; we encountered a problem once when we assumed that no *nfs* filesystems would be mounted in */tmp*, and found out about our mistake after *reaper* had removed all of the old files from a subdirectory which the Unix *rex* utility had mounted into */tmp*. Other daemons periodically shrink the size of system files such as */usr/adm/wtmp*. With this multitude of disk analysis and disk cleaning programs, our disk usage has been maintained at a reasonable level while the effects of full file systems have been minimized for us and our users.

CPU Monitors

Cpu loading is also monitored for potential problems. *Loadavg* analyzes *ruptime* output to determine which systems have a load average above a given threshold. We use the one minute load averages in order to maximize our responsiveness, but any average could be used. For systems with load averages above given thresholds it provides the top twenty cpu using processes. *Loadavg* runs from a central system and obtains information from *ruptime* and remotely executed *ps* outputs. It sends no output if system load averages are within reason. Ours is set up so that I receive mail only between eight and six on weekdays, and I get mail every five minutes for systems with a load average over ten, every fifteen minutes for load averages over seven, and every hour for load averages over five.

This is accomplished by having *loadavg* take a threshold as a parameter, and by making multiple cron entries with varying threshold parameters like so:

```
00,10,15,25,30,40,45,55 8-17 * * 1-5
                                /usr/local/bin/loadavg 10
20,35,50 8-17 * * 1-5
                                /usr/local/bin/loadavg 7
05 8-17 * * 1-5 /usr/local/bin/loadavg 5
```

Loadavg allows administrators to quickly assess cpu abuses and find runaway programs, as well as to find other abuses of cpu time. It facilitates better

balancing of loads across systems, and leads to quick detection of program and system inefficiency.

Hardware Monitors

Other resource analysis programs insure that needed hardware resources are definitely available. A system monitor daemon which we call *mstatd* informs system administrators within minutes after a system goes down thereby ensuring quick response to downtime. It frequently checks the status of *rwho* files and tests ethernet connections for old entries to determine that machines are down. Because it uses this method, it will reveal network outages as well as system downtimes. It helps us to maintain high standards of uptime for our systems. Since we have a lot of experimental and new hardware on our systems, it has proven to be a very valuable tool. *Mstatd* is written in C.

Another daemon, *killidle*, runs on some of our systems to kill off idle shells and keep terminal lines available for use. This ensures availability of ports into each system, which obviously makes users more comfortable and productive. *Modem.monitor* determines whether all modems on a system are in use. If all modems are being used it informs users who are on modems and playing games that resources are tight and they should stop playing. It also informs appropriate administrators who can determine whether or not to expand the modem system, or to better balance the modem needs of the systems. Working in parallel with disk daemons and *loadavg*, these programs have gone a long way towards keeping system resources available and well maintained.

Communication Monitoring Daemons

Mail Monitors

Another group of daemons probes our communication system for possible problems. Electronic mail is the main form of electronic communication among our employees, and the integrity of the mail system is therefore of the utmost importance. *Checkq.csh*, *lost_mail*, and *mailrpt* examine the mail system. *Checkq.csh* informs administrators of items which have been sitting in the mail queue (usually undeliverable for some reason) for an extended period of time. It only issues a report if such a condition exists, and it mails this report to responsible administrators. It runs on each system hourly. It is a simple shell script which analyzes *mailq* output.

Lost_mail reports on mail which has been delivered by mistake on the wrong system. It uses information from the *users* file along with remote listings to send warning mail to the owner of the account, and builds a report of all lost mail encountered which is mailed to the appropriate administrator. It runs at eight, noon, and four o'clock. This allows users to know of mail delivered on the wrong system, and allows administrators to know of problems associated with mail delivery.

These two programs provide confidence that a message which is sent gets to the person for whom it

was destined. *Mailrpt* runs weekly on each system to report the last time each mailbox was read and the last time the mailbox owner logged in. Administrators then peruse this report to determine which users may need education on how to use the mail system. Together, *Checkq.csh*, *lost_mail*, and *mailrpt* insure that internal mail reaches its destination reliably, and that that destination is prepared to receive it. With the reliability which our mail system acquires from the use of these daemons we can greatly encourage the use of electronic mail as an alternative to many oral conversations. When used properly, mail provides easy organization of information, flexibility in schedules and more time to address deeper issues.

Uucp Monitors

Two other daemons, *Update-Paths.csh* and *uusum* operate on our electronic connection to the outside world: uucp. *Update-Paths.csh* takes automatically delivered routing information and parses it into our *pathalias* routing database. The remote machine which delivers the maps to us keeps the time of the last map transmission and sends maps more recent than this time on to us when we poll them. *Update-Paths.csh* then installs these maps and calls *pathalias* to parse the data. It has almost eliminated time which our administrators used to spend doing this manually, and allows users to not normally have to worry about the connections necessary to route mail to its destination. Makefiles and shell scripts accomplish these tasks with a minimum os setup time.

Uusum provides administrators with daily summaries of uucp traffic and transfer rates. Administrators peruse this information daily to spot excessive or overly small transmissions and determine whether something is adversely affecting data transmission rates. Monthly outputs of *uusum* are archived and used to produce graphs of traffic for different classes of uucp usage, such as local, long distance, and *notes* sites. These graphs show growth patterns of *mail* and *notes* traffic. By analyzing mail traffic we have avoided many wastages of phone time, and have been able to better plan for future expansion. These two daemons ensure that our mail and uucp system is as easy to use and as adequate as possible.

Security Monitoring Daemons

The last group of programs which probe the system for potential problems address security. Setuid programs are monitored and approved through *suidaudit*, which provides daily reports on new or deleted setuid programs on each system. *Suidaudit* consists mainly of calls to the Unix utility *ncheck* which reports setuid files for given filesystems and a *diff* to produce the needed reports. By auditing setuid files, administrators have more control over root accesses on their systems. This decreases the number of often disastrous errors which users make while being root, ensures a central source of system administration, and prevents many cases of unauthorized access. The

system's failed login logs are perused daily for failed logins on dialups, and if such failed logins exist, it reports this fact to the appropriate administrator. This alerts the administrator to most attempts at unauthorized access to the system through modems. Since weekly account information is mailed to each user, *mailrpt*, discussed earlier, also helps security by providing information about stale accounts. By spotting unread mail, the administrator can find many accounts which are no longer used. Since the last login time of the owner of each system mailbox is included in the report, such analysis is very easy.

Mailrpt has led to the elimination of many unnecessary accounts from each of our systems. Another daemon, *pwchkr*, guesses user's passwords. It does so by guessing words and combinations of words from their finger entries in the *passwd* file, and words from a separate illegal password list (eg. we will not allow a password of "convex"). For most finger entry guesses it also asserts the word with the first letter capitalized. We use it internally to help protect ourselves from external and internal unauthorized access by informing owners of accounts with easily guessed passwords that their passwords should be changed. The first time we ran it it guessed about a fourth of our user's passwords. Most passwords guessed are either "convex" or the user name. These security daemons provide administrators with a great deal of information on areas of security which s/he should watch out for, and therefore help administrators to maintain security from inside and outside forces.

System Maintenance Daemons

Another class of daemons perform needed system maintenance. One such daemon runs nightly across all of our systems. This daemon, called *dopasswd*, creates accounts by generating new password files and needed login directories. It distributes some system files such as */etc/group*, */etc/hosts*, and part of */usr/lib/aliases*. It also performs needed yellow page makes and recomputes mail alias information. *Dopasswd* goes a long way towards creating a minimally standard computer environment and saves administrators a great deal of time in administering user accounts and system files. It can be run during the day to restandardize the environment or to create new accounts. It has saved us time in account creation and deletion as well as in chasing problems created by nonstandard environments.

Another maintenance daemon, *timed*, periodically synchronizes dates among our many machines. One machine acts as the master. That machine's date is kept accurate by an external clock corrected by radio transmission. This date is then used to synchronize remote machine times. These daemons attempt to ensure a consistent user environment across our machines, and thereby make employees feel more comfortable as they move among the systems.

Information Distribution Daemons

Yet another set of programs distributes system information to users and administrators. This information allows users and administrators to work more efficiently. Like programs which probe the system for problems, these programs have also enhanced user satisfaction with their computer environment and improved their productivity. Further, they have reduced the time I spend passing system statistics on to system users. Most of these information programs feed information into data files which are examined daily.

We use *notes* (news to many of you) for this purpose, since it is well structured for allowing administrators to sequence through new information while keeping old information around for later access. It also facilitates explanations of the information provided because of its capability to organize base notes and responses. We maintain a notesfile for each machine we administrate. The system error logs are stripped of unnecessary data and piped into a notesfile by a program called *geterrlog*. Although the system error log is different in format on different manufacturer's machines, from my experience there is always one present. Initial setup involves determining the location of the log and identifying extraneous messages which the log might contain. Error logs typically contain information on hardware failures, ethernet responsiveness, full file systems, bad spots on disks, and other operating system encountered errors.

By analyzing the data which comes from the system error logs without extraneous information, the administrator processes the information quicker, and is aware of many system problems which users might not report. *Hwconfig* and *swconfig* report daily changes to the system's software and hardware configurations. Changes are reported daily, and summaries are produced weekly. Configurations are posted on a bulletin board weekly so that users are more aware of the configurations on the systems on which they work. *Hwconfig* allows administrators to better troubleshoot hardware failures by providing information about what has recently changed. It also allows for correlation of system problems which might be dependent on revisions or existence of certain hardware. In a hardware development environment such as that of Convex, this information is very valuable to administrators as well as hardware development personnel.

Swconfig accomplished similar means with respect to software. System problems are more easily associated with the changed programs which may have caused them, and users can easily determine what software they are running under, and what is available on other systems. The *audit* daemon runs nightly on each system to determine system files which changed yesterday but were supposed to be unmodified from day to day. It produces checksums of system files, ignores files which are considered to be too volatile or otherwise unimportant in system audit,

and uses *diff* to determine the changes from day to day. This format allows administrators to process only needed information. The information allows us to track changed system files, to more easily associate problems with changes in the system, and to enforce one of our caveats that nothing goes into */usr/local/bin* without a *man* page and source. *Errlog*, *swconfig*, *hwconfig*, and *audit* provide us with the capability to know what the state of each machine was each day, to be knowledgeable on recent system changes, and to easily find problems introduced by the dynamic environment inherent to a hardware and software development organization. The history information which this system provides ensures that this valuable data will be available as new problems become apparent, and provides for better analysis of trends.

A final daemon, *phone.run.cron* produces and distributes electronic phone lists, allowing users to quickly retrieve logins and phone numbers of their fellow employees. Information is taken from the users file and from a static file describing phone numbers not associated with specific employees. Users can then use the *phone* command to get a phone number for a specific user. The phone number requester can provide the user name, first name, last name, or extension and can instantly retrieve the other information. *Phone* will also allow users to get phone numbers of conference rooms and remote sites. Further, *phone* allows users to access login and phone information without having to know which systems the user has accounts on. Because of this and the fact that it provides information on phone numbers which are not associated with a login, it is far more suitable than *finger* and *rfinger* for our purposes.

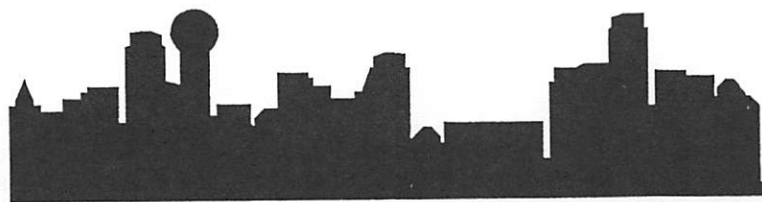
Future Plans

Plans are in place for the upcoming task of further expanding our administration daemon system. These plans provide for both more daemons and expansion of existing ones. We will attempt to build a bit more intelligence into our daemons allowing them to correct problems where possible and only report problems when they cannot be fixed. In doing so, a log will be required, but it will not be necessary to interrupt administrators with reports of easily fixable problems. This should make our administrators more productive.

The possibility also exists of creating an interface which allows administrators to better prioritize and respond to system problems quicker. The interface would contain such capabilities as a next error function, manual selections of predetermined types of responses to address standard problems, interruption if a problem arises which is designated to have a high priority, and a screen interface which displays a large amount of information simultaneously. With the tools we already in place, such an interface could be easily created. The expandability of such a system will be one of its primary design constraints.

Conclusion

All of the daemons which I have mentioned above combine to address many of the problematic areas of system administration which I have encountered. They provide me and my users with information about their system which makes them more comfortable and productive while working. The decrease in user-apparent problems which they provide also makes the users more comfortable and productive. They help to keep problem rates down, and keep problems small and easily addressable. They free administrators time so that it can be spent on development and other problems which require human intervention. By making the computer work for me I have accelerated in both the level of problems which I address, and the number of such problems which I can remedy, and therefore provide better service to the users I support.



Dave Taylor
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
taylor@hplabs.HP.COM

The Postman Always Rings Twice: Electronic Mail in a Highly Distributed Environment

ABSTRACT

Distributed high-speed LAN-based environments are interesting creatures, and one of the areas most likely to benefit from this is electronic mail.

Distributed Reception of Mail: Two unfortunate side effects of people having their own workstations are that whenever a message arrives the processing is likely to steal needed cycles from whatever task they're up to and that it becomes imperative that their machines always be up to receive mail.

Distributed Maintenance of Mailing Lists: One of the biggest complaints of administrators is that dealing with mail to `postmaster` is a real chore.

Distributed User Interfaces: One of the fundamental problems with the current model of electronic mail systems is that the various parts are very tightly coupled.

First time visitors to Los Angeles almost always ask why there isn't any sort of decent public transit system. There are a lot of reasons why Angelenos need to have their own wheels, including the socio-economic diversity of the region and the geographic sprawl of the LA Basin. The essential thing, however, and this is what seems to hit all transit schemes at one point or another, is that it

needs to be a well-thought-out system that extends throughout the *entire* trafficked region, not just one or two places.

This part versus whole dichotomy is exactly the same problem that we at HP Labs had with our electronic mail system. In a nutshell, we had a freeway, a few on- and off-ramps, and even a bus or two, but no *system*, no comprehensive package for the

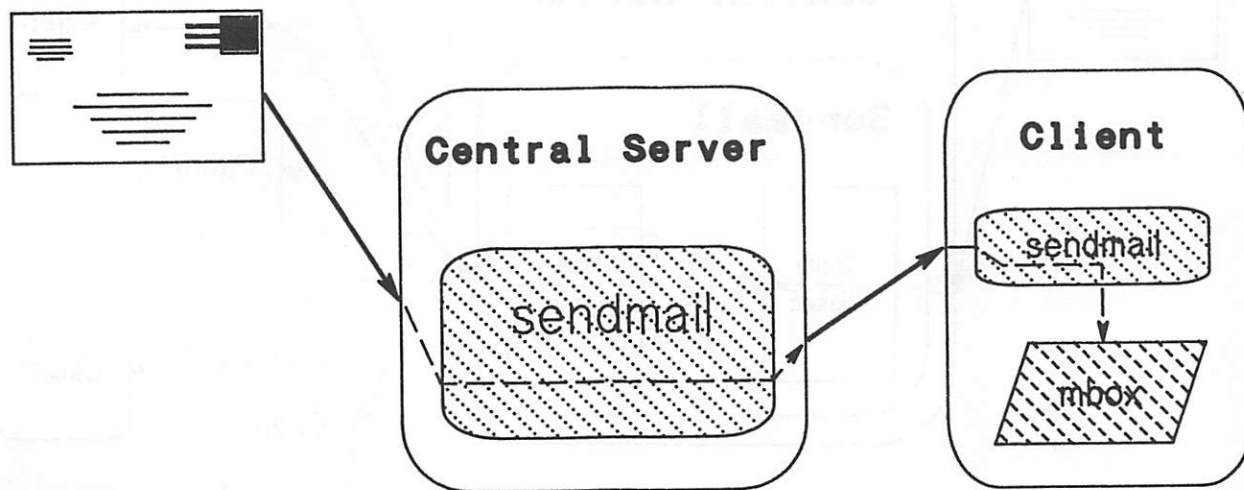


Figure 1.

installation, maintenance, and enhancement of mail in a highly distributed environment.

To remedy this, we analyzed "the big picture", looking first at how mail to a specific recipient got into our network and then traveled to its final destination; See Figure 1.

We also looked at the route taken by mail sent to internal redistribution lists; See Figure 2.

To further complicate the picture, users often switch off their workstations, change the access permissions to files or programs *sendmail* expects to find somewhere in their file system (a danger we'll return to later) or any of a number of other potentially dangerous changes to the overall mail environment. Even the central server isn't immune to fluctuations.

Through this analysis we identified three areas for enhancement:

Maintenance Of Mailing Lists — One of the most tedious positions in any electronic mail environment is that of "postmaster". Not only does postmaster get inundated with requests to join mailing lists, but also an amazing number of messages arrive daily asking simple questions like "what lists are there?" "What lists am I on?" "can you please add/delete me from a list?" and so on. With a lab of over 200 people, HP Labs postmaster was quite the most popular mail address.

To put the power in the hands of the people (with apologies to the Black Panthers) we created the Remote Alias Editing System: RAYS.

Centralized Mail Delivery was an intoxicating

idea: what if we could let people get mail when *they* requested a delivery, rather than forcing their machines to remain on-line and in a certain state?

This idea seemed to be achieved by the Post Office Protocol software shipped as part of the MH mail system (MH-POP), but, as we shall see, it ultimately failed to meet our requirements.

Dynamic Aliasfile Validation was the third area we targeted for improvement. When an administrator modifies the *sendmail* alias file, any related files, or any user touches their ".forward" file, the chances of *sendmail* violently rejecting the next message increases considerably. With over 2200 aliases, over 100 groups, and a passel of mail gurus about, the situation was unacceptably complex.

And so, out of chaos, *CheckAliases* was created. *CheckAliases* actually does all the work that the *newaliases* program ignores — traveling down all mail aliases, ensuring users exist, files are the correct mode, programs being invoked exist, and so on.

With the addition of these packages, HP Labs now has one of the most user friendly (I hate that phrase) Unix mail environments in the world, and we can only surmise what it will be like once we've brought up a PostOffice system . . .

We'll look at each area more closely, but before we do it's worth noting that our number one goal was to minimize the need for intervention by central administrators; instead trying to embody their knowledge and expertise in the software itself. This is the essence of good software!

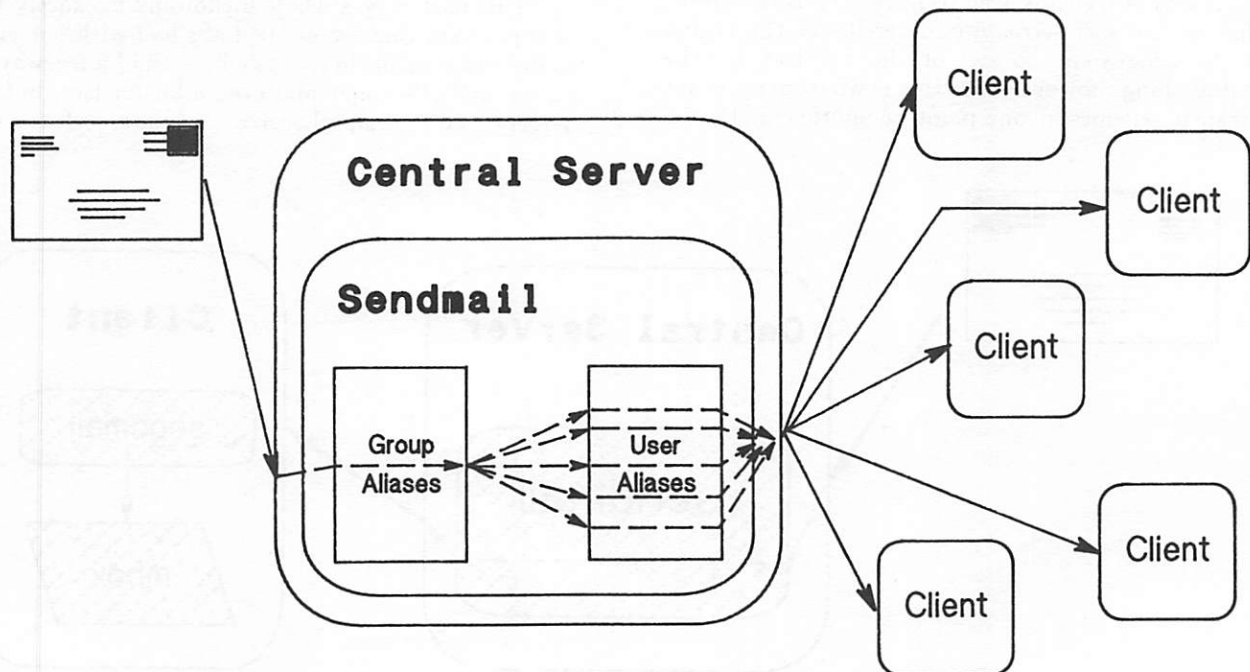


Figure 2.

The illustration in Figure 3 should give you a better understanding of the interrelationship of the various programs discussed herein.

Maintenance Of Mailing Lists

One of the most pervasive areas of electronic mail is that of distribution and mailing lists. Unlike their physical counterparts, addressed to "Homeowner at 400 Pine" or some other predetermined format, computer mailing lists are the electronic mail addresses of a group of people who are interested in discussing a specific area. Examples are the very popular "tcp-ip" mailing list discussing aspects of the TCP/IP networking protocol, "security" discussing computer security, and "Computers and Society", a forum for discussion of the impact of computers on society.

Within a complex environment like HP Labs, public list maintainers are usually instructed to include just one address for the site, with internal redistribution being done locally. Our organization also has private lists that reflect the centers, labs, departments, and projects within the Laboratory.

What this all adds up to is over 100 mailing lists accessible to HP Labs members. Our postmaster, already besieged with individual mail address delivery and maintenance problems, had their job greatly complicated with this diversity of lists, partly due to the added number of addresses involved (the lists have an average of 20 members, which is a total of over 2000 additional addresses) and partly due to the fact that it's always a 'moving target'. Analysis of the mail

sent to postmaster on a given week demonstrated that a significant percentage was dealing with four or five common mailing-list related topics:

- what lists are there?
- am I on this list?
- can you please add me to this list?
- can you please delete me from this list?
- can you please create list XYZ for me with the following members?

Notice that obvious extension requests like "what lists am I on?" were never sent since there was no easy way for postmaster to generate that information.

To eliminate these requests filling up the postmaster mailbox, we created the Remote Alias Editing System, or RAYS for short.

RAYS is based on the server/client model, with the two parts communicating via TCP. The server is started up from within the *inetd* network channel monitor program, and there is no limit to the number of clients that can be connected at any given time (practical limits were imposed in the interest of not overloading the server, however; the current limit is 15 simultaneous connections).

Before we delve further into the internals, let's just look at typical user session with the RAYS program. For this illustration, all user input is in bold face; See Figure 4.

This session shows a typical user finding out

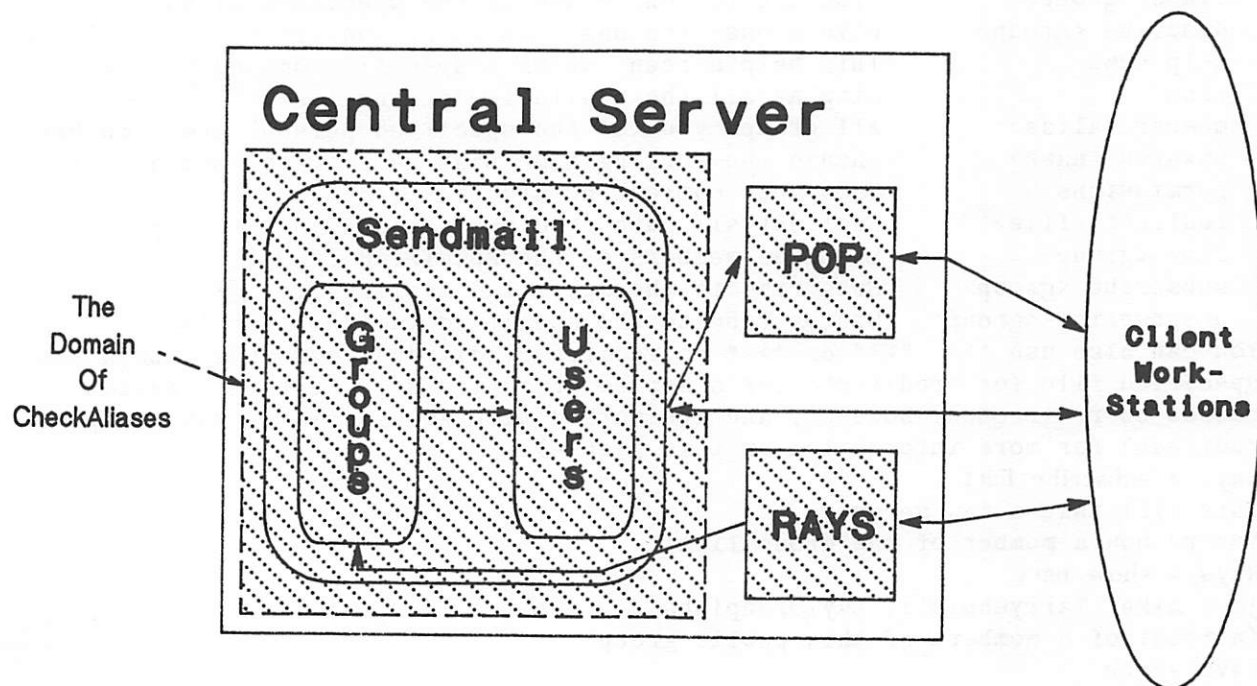


Figure 3.

what lists there are (those lists in parenthesis are private lists; users cannot *subscribe* to these lists from within RAYS), what the theme of a specific list is, what people are currently members of the list, and then choosing to add themselves to the list.

The interchange going on between the server and client is quite a bit more complex than this, however, and looks more like

user first brings up the RAYS program
server: HELLO hplabs.HP.COM,

I'm the RAYS server v3.0
client: LOGIN taylor
server: OK
client: STREAM
client: LIST
server: *a list of groups, spanning multiple lines*
server: .
client: DESC
server: *a list of group descriptions, spanning multiple lines*

% rays

Welcome to the Remote Alias Editing System, version 3.0

If you have any problems, please send mail to taylor@atom

You're connected as "taylor@hplabs"...

Getting the list of groups from the server...(read in 10 groups)

reading in group descriptions...(10 descriptions)

Rays > list

list1	list2	(list3)	list4
(list5)	list6	list7	list8
list9	list10		

Rays > describe list1

Group list1 is described as:

A group of people interested in the special properties of the number '1'

Rays > show list1

joe, mike, larry@hplplx

(a total of 3 members of this public group)

Rays > help

You're using the Remote Alias Editing System client program, version 3.0.

The commands available within the RAYS program are:

class <group>	List the current class of the specified alias
describe <group>	Give a one-line description of the group, if available
help <cmd>	This help screen, or on a specific command herein
list	Display all the available mailing lists
member <alias>	All groups you (or the specified person) are a member of
newname <name>	Change the working name that we're connected as. [*]
permissions	Show your current permissions [*]
redirect <file>	Redirect all output to specified file (or stop) [*]
show <group>	List the members of the specified group
subscribe <group>	Subscribe to the specified mailing list [*]
unsubscribe <group>	Unsubscribe to the specified mailing list [*]

You can also use ">", ">!" or ">>" to redirect individual commands into a specified file (or "redirect" for a series of commands). Certain commands cannot be redirected, however, and are marked with an asterisk. (see "help redirect" for more information on this please).

Rays > subscribe list1

This will take a few seconds...

You're now a member of the group list1

Rays > show list1

joe, mike, larry@hplplx, taylor@hplabs

(a total of 4 members of this public group)

Rays > quit

Figure 4.

```

server:
    now the user asks who's in list1...
client: MEMBERS list1 pub
server: joe, mike, larry@hplxp.hpl.hp.com
server:
client: TIMECHECK list1 pub
server: 43432433
    the user wants to add themselves — the
    system ensures that the version of the alias
    members list it has isn't out of date
client: TIMECHECK list1 pub 43432433
server: YES
    and then the system goes ahead and saves
    the modified alias
client: SAVE list1 pub
server: OK enter data, end with a '.' by itself
client: joe, mike, larry@hplxp.hpl.hp.com,
    taylor@hplabs.hp.com

```

The second type of logging is to a central log file; "/usr/adm/rays.log". The associated entry for the interaction above would be something akin to:

```

(taylor@hplabs.HP.COM) user login at
Tue Dec 29 15:21:00 1987
(taylor@hplabs.HP.COM) user subscribed
to group list1 (class public)
(taylor@hplabs.HP.COM) user left at
Tue Dec 29 15:22:17 1987

```

RAYS has a lot of other features, including the ability to have hidden groups (i.e. groups in the same general directory structure that aren't displayed to RAYS users), the ability to have users change the *name@host* that they're connected as, and a whole gamut of functions and commands available for list owners and superusers.

Centralized Mail Delivery

The second area that we were interested in was a system whereby we could offer a centralized delivery service, akin to the post office boxes available at most post offices. The computer analogue to the P.O. Box is called by the same name, and is associated with the ARPA Internet RFC-937 specification.

```

From: The Rays System (rays@hplms2.hpl.hp.com)
To: rays-events (People who keep track of RAYS)
Subject: taylor@hplabs.hp.com subscribed to group list1

```

The mailing list list1 has been modified by taylor@hplabs.hp.com.
The new list:

```
joe, mike, larry@hplxp,taylor@hplabs
```

Somewhat mindlessly,
- RAYS -

POP, as it's known, offers a protocol whereby client machines can query a central mail repository to find out what mail there is, and to request that messages be 'sent down the wire' to the client.

Typical POP installations, however, tend to be rather awkward, so to avoid this we set the following goals for our own implementation of a PostOffice system:

- It should be 100% reliable.
- It should be sufficiently fast that users have no noticeable delay receiving their mail at the point they explicitly request it.
- It should be sufficiently secure that unauthorized users cannot access postoffice boxes, yet authorized users may retrieve their mail from any host in the network.
- It should involve absolutely minimal administration, including adding users to the system, day-to-day administration and removing users from the system.
- It should be transparent to the user, or sufficiently easy to learn and use that it isn't viewed as an impediment to sending or receiving electronic mail.
- It should allow users to customize their interaction with the Post Office, including the ability to have mail automatically delivered, or to have mail requests emitted from an arbitrary application program
- It shouldn't require the use of a particular mail reader.

A tough set of criteria for a system, but a less-cohesive system wouldn't have been much help at all — it would have been ignored. client: . server: OK
finally, the user quits and the server immediately exits client: QUIT

Notice that as much work as possible is done on the client machine; the second request for lists, and both requests for group descriptions were handled completely on the client machine, as was the checking of the list to ensure the user wasn't already a member of the list they were requesting to join. One of the main RAYS design goals was to have as much work as

Figure 5.

possible done on the client machine rather than the server.

Another example of this distribution of work is the way that "member of" is implemented. When a user asks RAYS what mailing lists she is on, the client will use TIMECHECK and MEMBER requests to build a complete local list of all group members. As each list arrives, the client checks it for the specified user. If the user asks about membership in groups again, the program only needs to send the server TIMECHECK queries to ensure the local version of each list is current.

The RAYS server keeps a close watch on what goes on, with two levels of transaction logging. The first method is that the program sends electronic mail to "rays-events" whenever a group is changed, (amusingly, 'rays-events' is actually a mailing list that RAYS users can peruse). For example, the message sent from the transaction above would be something very similar to Figure 5.

In the world of POP systems, there is only one significant implementation of the RFC that is available to the public, and that is a part of the MH mail package. With misgivings, and after having done some research and prototyping of other ideas, we went ahead and brought up the MH-POP system.

By itself, however, MH-POP wouldn't have met any of the goals stated above. Fortunately we came across a program written by Ted Nelson of Fort Bragg, called *getmail*. *Getmail* is intended to be used as an alternative client program to the MH *inc* command.

The initial pass was to bring up the MH-POP server and the *getmail* client access program, but that was still insufficient for many of our design goals, most notably being the desire to avoid daily administrative overhead while offering a secure system. To remedy this, we enhanced the POP protocol and added significant features to the *getmail* program (indeed, ours is almost a complete rewrite of the program). The additions allowed users to see what mail they had without actually getting it, to specify what messages (by number only) they wanted to retrieve, to change their POP password from a client machine, and to add and delete 'aka's'¹.

With these additions, the POP user on their regular account could then alias 'from' to something like:

```
alias from 'popmail -f'
```

and easily check their incoming mail queue without even having to type in their postoffice box password.

The changes we made actually considerably enhanced the POP system to a point where it was more advantageous to use the *getmail* program than the MH *inc* command. For example, we could retrieve from the postoffice all messages from user 'carol' by

¹'aka' is FBI slang, of all things, for 'also known as', and in this particular instance refers to *user@host* pairs that are allowed password-less access to a postoffice box.

the following command:

```
getmail -m 'getmail -fn | \
grep carol | \
sed 's/:*//''
```

or could print out all our pending mail without removing them from our postoffice box with:

```
getmail -o - | lpr
```

and so on.

The other area not addressed by MH-POP was that of sending mail from programs other than MH; a user signed up for the POP service should be able to have her mail appear to be sent from their postoffice box rather than from the account she sent the message. This was a tough problem, but through some rummaging about in the *sendmail* configuration file and the addition of a simple address rewrite program on the client machines, we were able to have a simple system that users could enroll in to have their addresses rewritten appropriately.

[Why, I'm sure you're asking, didn't we just use the Reply-To: field, as that's what it's supposed to be used for? Well, as it turns out, the vast majority of machines in the real world don't parse or otherwise use that header, instead preferring to build their own return addresses based on routing or the From:/Sender: lines.]

With all these programs, scripts, and modifications in place, all available via our *ninstall* software installation system, the MH-POP system seemed ideal and ready to go.

That's when we started using MH-POP for large postoffice boxes full of mail, and postoffice boxes that had mail flowing into them constantly, and other permutations and found that our number one criteria wasn't met:

It should be 100% reliable.

While we realized that 100% reliability was probably a bit overly zealous as a goal, it wasn't unreasonable to expect it to work correctly 99% of the time. As more and more mail was being read, however, we found that the POP server was simply not reliable enough.

The first thought was that it was some subtle interaction or problem with the *getmail* program, but we tried accessing the same very large postoffice box with the MH *inc* command and continued to fail.

Three and a half weeks of trying to track down the problem later, we removed MH-POP from the server, and canceled the project. My final memorandum on the subject included:

After an extended period of running the MH-based POP server I am herein recommending that we terminate running the software and scrap any potential plans for using this particular system as a part of the supported HP Labs distributed electronic mail environment.

My reasons are as follows:

- *Most importantly - the POP server is not a reliable piece of software. It is all too common for people using the server to find that it 'arbitrarily' terminates their connection.*
- *As a result of trying to track that problem down - the POP server is based on code that is an integral part of the MH package, and that code is not only poorly organized and designed, but completely lacks any documentation, either as comments within the source code or as additional references.*

Considerable time and effort was spent in not only trying to offer a smoothly integrated POP environment, but also in fixing the conceptual and paradigmatic awkwardness of the original system...

...the problem with MH-POP notwithstanding, the experiment showed that the use of a system speaking a "post office protocol" would be a novel and helpful way of allowing people to deal with their email.

Our next foray into this area will be based on the CCITT X.400 '88 (also known as the Blue Book) central mailbox protocol specification, with the mail supported being in an X.400 format. We believe that this system, the ECMA Mail Server system, the PC Mail Protocol and related, more 'sophisticated', interaction specifications are the correct direction in which to be heading.

Dynamic Aliasfile Validation

One of the biggest headaches that continually plagued the administrative staff was the massive *sendmail* alias file they maintained on the central server machines. With over 2200 aliases in it, over 100 of which represented included files, 11 represented pipes to programs, and over 110 of which were explicit file archives, it was impossible to keep them all correct and up-to-date.

The problem was exacerbated by the quirky behavior that *sendmail* exhibited when confronted with any unusual situation. For example, programs being piped to don't need to be executable, but saving to a file that isn't mode '666' or greater causes the mail to bounce back to the sender, with a cryptic and unfriendly error message attached.

Certainly this wasn't a terrific state of affairs, and with the thousands of addresses we had, there was no way that our administrator could be sure that there weren't any typographical mistakes in any files, that none of the files being used as mail archives had been touched, that none of the programs being piped to or files being saved to had been deleted by some disk space fanatic, or even that users hadn't changed their account names.

Even more difficult were the problems inherent with ".forward" files; since those were under the control of the user yet were still governed by the same obscure rules as the aliases in the *sendmail* alias file. It was all too common for someone to say "gee, why

aren't I getting any mail?" and then to find out that they had done something 'unfortunate'.

In this chaos we looked closely at the behavior of the *newaliases* program and concluded that it did the absolute minimal work possible, simply taking each line of the *sendmail* alias file and breaking it into "alias" and "value", then adding the two fields to the blossoming DBM database. There were no checks for alias loops, non-existent files, included files that were in an unreadable format, or any other validation of the system.

The only check that *newaliases* does is to ensure that all addresses resolve down to one of the defined mailers on the system.

Further examination revealed that the problem with ".forward" files was even worse; people tended to make changes and then test it by sending themselves mail. If it got through then everything was terrific, and if it vanished then they assumed, quite correctly, that it was an indication that something was wrong.

In an attempt to collect all the necessary knowledge about the entries in the *sendmail* alias file, the environment around it, and the individual users ".forward" files, we designed and created *CheckAliases*. The *CheckAliases* program validates the DBM database created by *newaliases*, informing the user of any problems, possible dangerous situations, or anything else seemingly untoward.

The scenarios it checks for include:

- Addresses it cannot understand (an example of this was found on the very first run; we had an entry in the alias file of the form:

```
owner-hpux-windows::bwilliams
```

which, while it might appear acceptable, becomes a more obvious error when broken up into:

```
name = 'owner-hpux-windows'  
value= ':bwilliams'
```

The leading colon in the address made it an invalid address),

- Addresses in a users ".forward" file that cannot be locally delivered,
- Included files that don't exist any more (an example was seen as another obsolete alias, where the entry remained in the aliases file, but the actual included file had long since been deleted),
- Included files that aren't currently readable by the program,
- Included files that have non-address entries in them,
- Files destined as archives that either don't exist, or are in a mode that prohibits *sendmail* from successfully saving the message (this is probably one of the biggest hassles of the entire system),
- Pipes to non-existent programs, or programs that

aren't listed as being executable (this is more of a warning than an actual environment error as *send-mail* seems blissfully unaware of any potential problems),

- Bad individual addresses (typographical errors, accounts deleted, etc),
- ".forward" files that aren't owned by the right user, that have the wrong permissions, empty ".forward" files, or those with inappropriate entries,
- Users mailboxes with wrong ownership or wrong permissions.

As can be seen, *CheckAliases* does quite a comprehensive job of validating the entire mail alias database, checking each entry as it goes along.

To report its findings, the program defaults to simply listing errors to standard output, but there is also an option to have the output mailed to a pre-specified address in a nice, readable format; See Figure 6.

Remember that the old way of finding these mistakes would have been for some innocent user to have

sent mail to, say, "cl-objects", and then to have it bounce back because "mayerhplnrm" wasn't an acceptable address (and then for the user to most likely then throw the error away — the most common reaction — or to assume no-one on the list received the message and then send it again).

Finally

We've demonstrated that the Unix electronic mail environment can be considerably improved, taking it from being rather unfriendly and awkward to being a system where even neophytes should be able to keep their environment at top-notch performance and efficiency. This newly organized environment has absolute minimal failed messages, and as much power as possible in the hands of the individual users of the system.

We'd be very interested in hearing about other research and development done in this area - please feel free to drop me a note either via electronic mail or, if that doesn't work, via paper mail.

```
From: The Alias Check Program <postmaster@hplabs.HP.COM>
To: taylor@hplabs.HP.COM (Dave Taylor)
Subject: problems with hplms2 alias file
```

```
For host "hplms2":
```

```
Warning: I encountered an address I couldn't understand.
(for mail alias owner-hpux-windows --> ":bwilliams")
```

```
Mail to alias "bboard" is supposed to be piped to a program,
but I can't find it ("/usr/local/lib/news/recnews")
```

```
Alias "texhax-hp" has an :include: for a non-existent file:
("/usr/local/lib/distribution/texhax-hp")
```

```
Mail to alias "hp_sites" is supposed to also save to a file,
but I can't find it ("/usr/local/lib/mail/maps/autoroute")
```

```
Mail to "YES" is not locally deliverable. This address can be found
listed in the following alias or aliases:
("aiws")
```

```
Mail to "mayerhplnrm" is not locally deliverable. This address can be found
listed for the following alias or aliases:
("cl-objects", "cl-windows", "test-alias")
```

These 6 problems were encountered while checking 1069 aliases and 1712 addresses.

Automatically, but Sincerely,

-- The Alias Check Program

Figure 6.

One final note about availability of the sources;

Hewlett-Packard Labs is a research organization. We have no direct product responsibilities. As such, we cannot determine what products are available or when they are available. However, we occasionally offer specific software packages to the general public, so please contact us if you are interested in obtaining any of the software discussed herein.

References

WAYS

- [Birell, A. D. et. al.] *Grapevine: Two Papers and A Report*, CSL-83-12, Xerox Palo Alto Research Center, December 1983.
- [BITNET List Server] informal electronic mail, 1987
- [Kincl, Norm, et. al.] *Managing a Distributed Environment*, EUUG Conference Proceedings, Autumn 1986.
- [Taylor, Dave] *The WAYS Administration Guide and Communications Protocol Specification*, HP Labs Internal Memorandum, August 1987
- [Taylor, Dave] *The WAYS User Guide*, HP Labs Internal Memorandum, August 1987.

POP:

- [Borenstein, Nathaniel, et.al] *The Andrew Message System - A Portable Distributed System for Multi-Media Electronic Communication*, Carnegie-Mellon University, 1986.
- [Butler, M. et. al.] *Post Office Protocol - Version 2*, SRI Network Information Center Request-for-comments #937, UCS-ISI, February 1985.
- [Clark, David, et. al.] *PCMAIL: A Distributed Mail System for Personal Computers*, SRI Network Information Center Request-for-comments #993, The Massachusetts Institute of Technology, December 1986.
- [Crocker, David] *Standard for the Format of ARPA Internet Text Messages*, SRI Network Information Center Request-for-comments #822, The University of Delaware, 1982.
- [Jin, Tai] *Ninstall: A Software Distribution Package for Networked Environments*, (draft copy), HP Labs Internal Memorandum, 1987.
- [Jin, Tai, et. al.] *Network Based Software Distribution*, HP Labs Software Technology Laboratory Technical Report, Dec 1987.
- [Nelson, Ted] comments within the *getmail* program, Fort Bragg/SRI International, 1987.
- [Rose, Marshall] *Post Office Protocol (revised), Extended Service Offerings*, a draft request-for-comments, Northrop Research and Technology Center, December, 1985.
- [Rose, Marshall] *The Rand MH Message Handling System: Administrator's Guide*, The University of California, Irvine, 1986.

[Rose, Marshall, and Romine, John] *The Rand MH Message Handling System: Users Manual*, (UCI Version), The University of California, Irvine, 1986.

[Taylor, Dave] *A Guide to the Installation and Administration of the MH-POP System*, HP Labs Internal Memorandum, June 1987.

[Taylor, Dave] *A Users Guide to the MH-POP System*, HP Labs Internal Memorandum, June 1987.

[Taylor, Dave] *HP-POS - The Hewlett-Packard Post Office System*, HP Labs Internal Memorandum, June 1987.

[Taylor, Dave] *The Elm Mail System Users Guide*, Hewlett-Packard Laboratories, 1987.

[Taylor, Dave] various electronic mail messages, dated August 1987, mostly addressed to Norm Kincl and Ted Wilson, both of HPL.

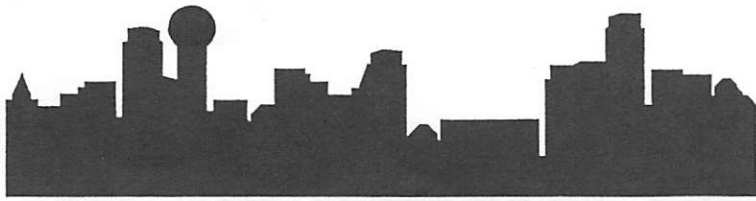
CHECKALIASSES:

[Allman, Eric] *Sendmail Installation and Operation Guide*, Britton-Lee Inc, July 1983

[Allman, Eric] *Sendmail - An Internetwork Mail Router* (draft copy), Britton-Lee, Inc, July 1983.

Conversations with Karen Bradley, Jeremy Levish, and Mike Rodriguez, administrators of the HP Laboratories Computing Environment.

[Taylor, Dave] *Mail Aliases in a Distributed System - The CheckAliases System*, HP Labs Technical Memorandum, December 18th, 1987.



Joseph N. Pato
Elizabeth Martin
Betsy Davis
Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824
617-256-6600
...{yale,mit-eddie,decvax}!apollo!pato

A User Account Registration System for a Large (Heterogeneous) UNIX Network

ABSTRACT

Three problem areas arise when considering a user registration system for a large heterogeneous distributed computing environment. Large environments demand controls on the complexity of administration. Heterogeneity requires an examination of the notion of identity in the network as well as the interoperability of software on different hosts. Distribution raises the problems of availability, reliability and security.

Generally available UNIX environments (bsd4.3 and AT&T SYS5.3) provide few tools for solving these problems. Account administration is typically handled through manual editing of a single `/etc/passwd` file. Consistency is maintained on multiple machines by periodically copying the `/etc/passwd` file to each machine in the network. For large networks with thousands of users and machines, these mechanisms are clumsy and error prone, and they vest too much power in a single system administrator.

RGY is a replicated user registration system built on Apollo's portable Network Computing System (NCS). The system consists of a set of daemons which maintain a replicated user registration database. Remote access to the user registration database is provided at each client site through remote procedure calls in a portable subroutine library that replaces the `getpwent(3)` and `getgrent(3)` C library calls. Weakly consistent replication provides a high degree of availability and reliability. Propagation of individual updates is performed yielding an inexpensive mechanism for maintaining consistency. Updates are securely performed using authenticated interfaces, allowing any client site to update the database.

Introduction

In a conventional single host UNIX environment, system account administration is managed through manipulation of the `/etc/passwd` and `/etc/group` files. Generally a system administrator is responsible for properly editing these files as well as performing the system house cleaning associated with the arrival and departure of users. When UNIX was primarily an operating system associated with departmental mini-computer environments that consisted of few (under 100) users, the burden on the administrator was tolerable.

In the mid 1980's, networks of inexpensive UNIX workstations became common, providing a workstation owner with a high degree of autonomy, as well as guaranteed response time in the absence of time-sharing. As long as each workstation or cluster of workstations remained autonomous, the administrative burden associated with each machine remained low. Account management could be delegated to

members of the user community for each workstation or cluster.

With this autonomy, however, early workstation users also encountered isolation. Data and resource sharing became cumbersome, relying on bulk data transfer protocols like FTP and virtual terminal protocols like Telnet. To recover some of the cooperation found in time sharing systems, computer vendors introduced distributed file systems like Apollo's DOMAIN [Leach 83], Sun's NFS [Sun 86] and AT&T's RFS [Rifkin 86], and later developed network computing environments, like Apollo's Network Computing System (NCS) [Dineen 87] and Sun's ONC [Sun 87]. Network computing environments provide heterogeneous compute and resource sharing while distributed file systems provide data sharing. Distributed file systems can be considered a subset of network computing environments. Therefore, for the purposes of this paper, we will use the term network computing environment to refer to either of these forms of network resource sharing.

A network computing environment transforms a network of workstations from independent administrative jurisdictions to a federation of loosely coupled systems. For access control mechanisms to be meaningful, every system must share a single representation for users' *credentials* (user names and user IDs). With independent workstations, the assignment of user names and user IDs needs to be unique only on each machine; in a network computing environment, this assignment must be unique across the network.

Since a user's credentials must be unique across the network, system administrators can no longer delegate account management responsibilities to individual workstation user communities without compromising the security of other user communities in the federated system. In contrast to isolated workstations, which can diffuse the administrative burden of account management, a network computing environment forces account management responsibilities to be assumed by a network administration authority. This network administrator accumulates the requirements of each workstation user community and then must redistribute the account information to each workstation in the federation.

RGY, a replicated user registration system built on Apollo's NCS, has been developed to allow the administration of large network environments. Our goal is to provide a network user registration system that will work well in a network of tens of thousands of hosts and users. To accomplish this we have developed the replicated user registration database, *RGY*, to serve as the secure repository for network system account management information. Access to the *RGY* database is provided through NCS remote procedure call interfaces exported by a collection of daemon processes known as *RGYDs*.

Hosts that wish to participate in the federated system access the *RGY* database through existing `getpwent(3)` and `getgrent(3)` C library calls as well as through additional query and update primitives. Each host is the final authority in granting access to its resources. The *RGY* database allows the federated hosts to provide a consistent view of the user community, but each host is free to filter the information from the *RGY* database to restrict access, or to correct for differences in the local file system.

Existing mechanisms for coping with network account administration are briefly examined in section 2. Section 3 describes the data model and tools we developed to allow for division of labor in maintaining the user registration database. Section 4 addresses the issues of providing highly available, reliable and efficient access to the *RGY* database, and Section 5 covers the mechanism to secure update access to the *RGY* system.

Existing Systems

Password file maintenance is a real and present problem for administrators of large UNIX installations. The 1987 USENIX Large Installation System Administrators Workshop drew numerous position papers on UNIX account management and the distribution of information across networks. At this workshop, attendees discussed the evolutionary processes that result in large installations and the need to unify account information in the resulting network [Cottrell 87]. Other attendees described the use of structured editors for adding entries to the password file and the subsequent semi-automatic copying of the file to all hosts in the network [Leeper 87]. These current approaches, centered around the existing UNIX data and administrative models, are cumbersome in today's small networks (fewer than 100 hosts) and hold little promise for the large (thousands of hosts) networks of the future.

Remote access to the login account database allows replication strategies to limit their focus to a strategic subset of the network. Sun's Yellow Pages (YP), part of ONC, is a simple network lookup service that has been used to provide remote access to password file information. While YP did not modify the `/etc/passwd` data model it did introduce the use of remote procedure calls to remotely access the UNIX login account database.

Outside the UNIX environment, the XEROX Grapevine system [Birrell 82] has addressed many of these issues. Grapevine was intended to be primarily used as a delivery mechanism for a large, dispersed computer mail system. It did not directly address the issues of maintaining a replicated user login account database. It maintained a replicated database of mail users which, in some instances, could also be used for user logins. Its concern for the issues of authentication, access control, decentralized administration, replication and scalability has served as inspiration for much of the work described in this paper.

Administrative Model

Large computing environments tend to contain a large number of both users and machines. Frequently these consumers and resources span administrative or organizational domains. To accommodate this type of environment we have enhanced the UNIX identity model to include the notion of organization. In addition, we have added access control objects to each entry in the user registration database. These changes allow mutually suspicious system administrators to cooperatively manage a logically partitioned user registration database. Administrative complexity is further reduced through the use of a structured editor for database manipulations.

The RGY Data Model

The RGY user registration system maintains a database consisting of *naming* information for people, groups and organizations, *login account* information for people, and general system *properties* and *policies*. In the */etc/passwd* file people and accounts are combined in a single record. We, however, feel that people and accounts are distinct objects in a user registration system: accounts represent active roles that people can play when accessing the system, whereas people maintain passive roles through the ownership of files, receipt of mail, etc. that persist independent of the existence of an account.

Groups and organizations are collections of people. Groups retain their conventional UNIX semantics and exist to allow a collection of people to share privileges to system objects. Organizations provide another dimension for sharing. Apollo has extended the UNIX file protection model from user, group, others (rwxrwxrwx) access to user, group, organization, others access. Organizations can be used just like groups, where a person can be a member of any number of groups. More typically, however, we use organizations as a means of partitioning the global user community into administrative jurisdictions, where each person belongs to a single organization.

In addition to maintaining information about the users and logical groupings of the networked system, the RGY database contains system policy information. Policy information consists of system configured minimum password length, password content restrictions, password expiration lifetime, absolute password expiration date, and account lifespans. Policy is never enforced by the user registration system. It exists as a guide for clients of the system.

Naming Information

The naming database is divided into three relations, also referred to as naming domains, that establish the existence of individual persons, groups and organizations within the registry. An entry in one of these naming domains is called a *PGOitem*. A *PGOitem* establishes the binding between a name and a set of credentials which consist of a unique identifier (UID) and a *unix id*. The *unix id*, preserved for compatibility with password file entries, is a small integer value used as a user id for people, a group id for groups and an org id for organizations. Aliases, multiple names mapping to the same credential information, are allowed to exist.

PGOitems contain a *fullname* field, an *owner* field and miscellaneous properties. A *PGOitem* can contain a list of typed mail data. This data consists of a type code and an uninterpreted *printstring*. The *printstrings* may be interpreted by a system mailer and usually define the preferred delivery mechanism or mailbox to be used.

Groups and Organization *PGOitems* have associated membership lists/ A membership list enumerates the people that have the rights and privileges of the

group or organization. Organization *PGOitems* may also contain policy information. By establishing policy, an organization may impose stricter password and account discipline than the other organizations in the registry. The actual policy data for an organization can be retrieved for editing, but most operations that yield policy information return the *effective policy* data. To determine an organization's effective policy, the system compares the organization policy information with the base registry policy and returns the most restrictive value for each field.

Login Accounts

Accounts contain a superset of the information stored in the */etc/passwd* file. An account entry is divided into two portions. The user portion of the account contains the home directory, login shell, password and *gecos* fields. The administrative portion contains information about the creator of the account, account expiration date and other information to indicate the validity of the account. An account defines a subject identifier (SID). A SID is a UID triplet which identifies the person, group and organization that correspond to the account.

UIDs [Leach 82], which are used extensively throughout the Apollo system, are a 64 bit concatenation of the current time and host network address. Unlike the *unix ids* which are assigned by the system administrator, *UIDs* are generated for the *PGOentry* by the RGY system and are guaranteed to be unique.

Accounts are keyed by login name, which is the concatenation of the person name, the group name and the organization name separated by periods (e.g., the user *smith* might have the account *smith.sys.r_d*). Login names can be abbreviated; accounts define the minimum abbreviation necessary for their selection. In the example above, the account *smith.sys.r_d* could be accessed as *smith.sys* if the associated abbreviation was person and group, or as *smith* if the associated abbreviation was simply person. Each person may have multiple accounts, either by using aliases, or by creating accounts with different abbreviations.

Decentralized Administration

Owner fields in *PGOitems* and registry properties allow mutually suspicious system administrators to securely partition the administration of the RGY database. An owner field defines who can update the corresponding record.

Access Controls

The RGY database maintains certain access controls when updating information in the database. Only the registry owner, stored in the registry properties, can update the registry properties or policy. The registry properties also contain owner records for each of the naming domains. Only the owner of each naming domain can create new *PGOitems* in that domain.

When a *PGOitem* is created, it is assigned an owner. All future manipulations of that *PGOitem*

can only be performed by the owner. Group and organization membership lists can only be manipulated by the owner of the group or organization PGOitem.

Accounts can be created only by the owner of the corresponding person PGOitem. To have an account that is affiliated with a specific group and organization, a person must first be a member of the corresponding group and organization. Update of the administrative portion of accounts is reserved to the owner of the corresponding person PGOitem; updates to the user portion of an account can only be performed by the corresponding person, or by the owner of the corresponding person PGOitem. If a person is deleted from a group or organization membership list, then any accounts that may exist for that person in the group or organization are also deleted. Group and organization membership as a pre-condition for account existence is an invariant that is maintained by the RGY database.

The Representation of Owners

An owner field is represented by a SID where any constituent field may be replaced by a wildcard (represented by the character '%'). Owner permissions are granted to anyone who can login with an account that has a SID that matches the owner SID. If the owner SID contains a wildcard for one of the constituent fields, then any value for that field will match. In a free-wheeling system, all the owner fields could be set to %.%., thus allowing anyone to manipulate all the data in the RGY database. Owner records of the form %.rgy_admin.% would grant permission to anyone logged in with an account that had rgy_admin as the group portion of its SID.

If all owner fields in the RGY database are the same, then update access to the RGY database is comparable to update access to the /etc/passwd file. When the RGY database contains a large number of people, however, it is more likely that each user community will have the PGOitems corresponding to its members owned by an administrator from within that user community.

Example

For the purposes of this example, we will assume that the network and machines in the federation are secure. The only security risk we are concerned with is the access to or corruption of data by a person with a valid but unauthorized account.

In a large corporation, a small group of researchers are working on a sensitive project called the manhattan project. To protect the confidentiality of their work, they have protected their files so that access is limited to members of the manhattan group. The researchers could disconnect their machines from the corporate federation and insure their security, but to do so would unduly disrupt their work. How do they guarantee that no one outside the group acquire an account with access to their data?

The first step is to assign the ownership of the manhattan group PGOitem to a member of the manhattan group (e.g., teller.manhattan.research). This will guarantee that only the user teller, who is a member of the manhattan group can add or delete members from the group. For most situations this will be sufficient, but for truly security conscious environments more must be done.

Assume that the user fermi is a member of the manhattan group. Further assume that the owner of fermi's person PGOitem is malicious.spy.%. It would be a simple matter for malicious to change the password on fermi's account and thus compromise the manhattan group's data. To be fully secure, the administrator of the manhattan group (teller) should allow people to be members of the group only if he is also the owner of their person PGOitems.

Structured Editing

The edrgy tool is used to manage the naming, account, and policy information in the RGY database. It is an interactive editor that provides users and system administrators with a structured interface to the user registration system, at once ensuring consistency, semantic correctness, and timely availability of changes.

Edrgy is aware of the semantic constraints placed upon the contents of the RGY database, and of the policy that is in effect. Edrgy uses this knowledge to assist the system administrator in performing semantically correct operations. For example, if the system administrator attempts to add an account for a person that does not belong to the requested group, and the administrator has rights to update the group, then edrgy will first add the person to the group and then add the account. Warnings are given before an operation is performed if that operation may have side effects. For example, edrgy will warn that the deletion of a group will also delete any accounts that exist with that group's permissions.

System Structure

The RGY user registration system is composed of two distinct portions: the database, which is an NCS replicated object, and the client agent which provides RGY access for the host environment.

RGYD: the RGY daemon

RGYD is the NCS server (process) that exports remote interfaces to the RGY database. Three classes of interfaces are exported by the RGYD: database queries and updates, replica control, and database update propagation.

Database operations

Database operations involve the maintenance and use of the RGY database. RGYD exports interfaces to query and update all RGY structures directly. In addition, the RGYDs maintain a set of interfaces presenting a view of the database that is equivalent to

the view presented by the `getpwent(3)` and `getgrent(3)` C library functions. By extracting information from the corresponding PGOitem and account records RGYD constructs password file entries. Group and org file records are constructed from the corresponding PGOitem and membership lists.

The RGY database is kept in virtual memory as a forest of balanced binary trees [Aho 74] yielding efficient ($O(\log n)$ operations where n is the number of items in each relation) query and update access. Deleted items are marked and left in the trees until garbage collection is performed during a checkpoint. Updates are first applied to the in memory data structures and are then atomically recorded in a stable storage log. Checkpoints of the in-memory data structures are taken every few hours for each relation that has been modified since the last checkpoint. The RGYD automatically recovers the state of the RGY database after a system crash by reloading the last checkpoint state and then re-executing each operation recorded in the stable storage log.

Not all UNIX programs access the password file structures through the procedural interfaces provided by the C library. To accommodate these programs, the RGYD maintains ASCII file versions of the password, group and org file. These files are recreated at each checkpoint if the data in these views has been modified.

Replica Management

A collection of RGYD processes spread across a number of hosts cooperate to maintain a weakly consistent replicated database. Updates do not occur at all RGYDs simultaneously; instead, one of the RGYDs is selected to serve as the master site and becomes the only daemon that accepts database updates. The master RGYD then assumes the responsibility of propagating each update to the other cooperating (slave) RGYDs.

RGYD sites may come, go and move around with ease. When a slave RGYD first starts running, it locates the master RGYD through the NCS Global Location Broker and announces its existence. If this RGYD is a new site, then the master RGYD will initialize the slave and record an operation to inform all other slaves of its existence. If the new RGYD is an existing site that has moved to a new address, the master RGYD will record this change of address and inform the other replicas. In this way each RGYD maintains a current copy of the replica list.

A special tool, *rgy_admin*, is used to remotely inspect and control each RGYD. All operations that affect the state of a RGYD are reserved to the owner of the registry database as recorded in the RGY properties data. With the *rgy_admin* tool, the registry owner can determine if replicas are out of date, cause a replica site to be reinitialized, select a new master site and decommission a RGYD site. When a RGYD site receives the decommission request, it purges its database and terminates execution.

Update Propagation

In addition to managing the database and replica list, the master RGYD also manages a propagation queue. The role of the propagation queue is analogous to the stable storage log. Every update operation performed at the master RGYD is recorded in the propagation queue for later application at the slave RGYDs. In practice, the propagation queue and the stable storage log are the same structure. Slave RGYDs are free to truncate the stable storage log once a checkpoint has completed, but the master RGYD must preserve the portion of the log that remains to be propagated to each slave. Update propagations, like all other remote operations, are accomplished through the use of a remote procedure calls.

A simple protocol between the master and slaves insures that updates are processed in serial order. The master RGYD applies a monotonically increasing timestamp to each update it records. When propagating an update, the master RGYD transmits the previous update timestamp as well as the current update timestamp. Retransmitted updates are simply ignored by the slaves, but if the slave detects that it is out of date with respect to the previous update it requests to be reinitialized.

The master RGYD periodically retransmits an update to a slave which is unreachable. As the number of attempts to reach the slave increases, the time interval between retransmissions is also increased. Eventually the master will mark a slave as out of touch and will re-initialize the slave when it finally becomes reachable. Database initialization is accomplished through bulk transfer of the database state to the target slave RGYD. Thus the master may purge updates from its propagation queue even when some slaves are unreachable for long periods of time.

The RGY Client Agent

The RGY Client Agent (RCA) is divided into two components. The most primitive level consists of the automatically generated client side RPC stubs for the registry operations and code for binding to a RGYD. The next, optional layer of the client agent (RGYC) provides registry services in the event of network failure. This layer can be used for translating credentials in a heterogeneous environment or for filtering data from the network registry.

The first time a RGY operation is performed, the RCA contacts the NCS Global Location Broker to randomly select a RGYD site for the operation. Subsequent operations are directed to the same RGYD until that server becomes unavailable. To allow the client agent to remain unaware of the replication strategy chosen by the RGYDs, we have divided RGY operations into separate query and update interfaces. The RCA actually maintains two bindings, one for queries and the other for updates. With the master/slave replication currently implemented by the RGYDs, only one server will register the update

interface at a time. An explicit binding operation is provided by the RCA for registry editing applications. This allows the application to force queries and updates to be delivered to the same RGYD.

The Local Registry

The local registry, maintained on each node by the RGYC, provides a cache of user registration data in the event that a registry server is not available. This cache of recently used accounts supports queries for login and C library calls (`getpwent(3)`, `getgrent(3)`). In order to prevent the cache contents from becoming stale, each remote operation returns the timestamp of the last operation that may have invalidated the cache. If the cache is out of date, the client agent initiates a cache refresh operation.

Authentication

Ignoring well known security holes in UNIX, it can be said that access to files is vigorously protected by the operating system. When deciding to grant access to a file, the kernel is free to believe the identity information that it has stored for the process. In a network computing environment, however, there is no reason for one host to believe that another host has not been compromised. An application cannot even be sure that network messages truly originated with the host listed in the message.

The traditional mechanism for proving identity is the use of a secret that is known only to the two principals, the person claiming the identity and the guardian of the resource. In a network environment where principals reside on different hosts, encryption must be used when exchanging the secret in order to maintain the secret. Our model for network authentication is inspired by Needham and Schroeder's work [Needham 78].

All RGYD update and replica administration operations that apply access controls perform authentication as the first step in those controls. To perform authentication, two encryption keys are associated with each account: a login key which is constructed from the plain text of the account's login password, and a master key which is generated by the RGYD when the account is created. When an update request is received by the RGYD, it constructs a random bit pattern and encrypts it with the requester's master key. The RGYD then makes an RPC callback to the initial requester. This callback is a challenge that requires the requester to decrypt the message, perform a function on the bit pattern, and return the encrypted result.

To successfully meet the authentication challenge posed by the RGYD, the requesting process must possess the valid master key for the claimed identity. The login (`/bin/login`) and set user id (`/bin/su`) programs have been modified to acquire the valid master key. Rather than using the standard `getpwent(3)` calls to retrieve the password file record, these programs now make a direct RCA call that

retrieves the required information as well as the master key. To protect the master key, it is never transmitted in the clear. It is first encrypted with the login key for the account. In this way the master key will be useful only if the login program possesses the valid password for the account. If the password is not known, the master key will not be decrypted properly.

The mechanism described above is used to guarantee that the RGY database is never modified by unauthorized users. In a security conscious environment, it is also necessary to verify that the client is connected to a legitimate RGYD. Mechanisms to accomplish this task are inherent in the system, but are beyond the scope of this paper.

Conclusions

The RGY system is currently used by about 100 personal workstations on the Apollo corporate internet. The RGY database contains about 2500 users, 100 groups and 50 organizations. On an average day about 150,000 database operations are performed spread out over the 4 RGYDs maintaining the replicated database. While these numbers are small compared to our design goals, we are encouraged to see that we are not yet close to saturating the capacity of a single RGYD even when only one server is running.

We are currently investigating new replication algorithms that will allow us to perform updates at any RGYD site, rather than at only the master RGYD site. These algorithms maintain the semantic invariants in the database, and will improve update availability in the face of network failures.

References

- [Aho 74] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *The design and analysis of computer algorithms*, Addison-Wesley, Reading Ma.
- [Birrell 82] Andrew D. Birrell, Roy Levin, Roger M. Needham and Michael D. Schroeder. Grapevine: an exercise in distributed computing. *Communications of the ACM*, Vol 25, No. 4, April 1982.
- [Cottrell 87] Pete Cottrell. Password file management at the University of Maryland. In *Usenix Proceedings of the Large Installation System Administrators Workshop*, 32-33, April 1987.
- [Dineen 87] Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, Geoffrey L. Wyant. The network computing architecture and system: an environment for developing distributed applications. In *Proceedings of the Usenix Association Summer Conference*, 1987.
- [Leach 82] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton and Paul H. Levine. UIDs as internal names in a distributed file system. In *Proceedings of the Symposium on Principles of Distributed Computing*, 34-41, Association for Computing Machinery, 1982.
- [Leach 83] Paul J. Leach, Paul H. Levine, Bryan P.

Douros, James A. Hamilton, David L. Nelson and Bernard L. Stumpf. The architecture of an integrated local network. *EEE Journal on Selected Areas in Communications*, SAL-I(5):842-857, 1983.

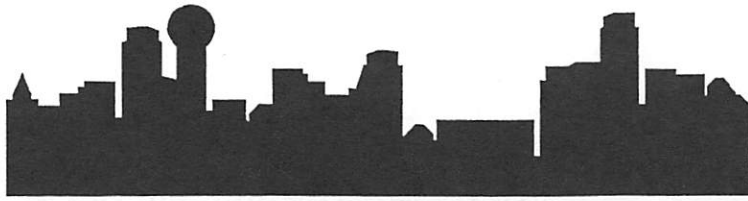
[Leeper 87] Evelyn C. Leeper. Login management for large installations. In *Usenix Proceedings of the Large Installation System Administrators Workshop*, 35, April 1987.

[Needham 78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, Vol. 21, No. 12, 993-999, December 1978.

[Rifkin 86] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, Kang Yueh. Remote file sharing architectural overview. In *Proceedings of the Usenix Association Summer Conference*, 248-259, 1986.

[Sun 86] Sun Microsystems Inc. Networking on the Sun workstation. Part no. 800-1324-03. 1986.

[Sun 87] Sun Microsystems Inc. Open network computing technical overview. DE240-0. 1987.



USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

Project Accounting on UNICOS

Charles K. Eaton
General Electric Company
NAS Integrated Support Contract, MS 258-6
NASA Ames Research Ctr.
Moffett Field, CA 94035
ames-prandtl.ARPA!eaton

ABSTRACT

NASA's Numerical Aerodynamic Simulation (NAS) Program and Cray Research Inc. (CRI) have developed new elements extending UNIX for a supercomputing environment, which are now incorporated in Cray's UNICOS operating system. We at NAS modified AIM Technology's *Job Accounting* to work with UNICOS and to meet our project accounting requirements. The resulting systems provide many types of usage reports for resource management and control.

INTRODUCTION

The first implementation of UNICOS (CRI's UNIX-based operating system) project accounting software was done here at NASA's Numerical Aerodynamic Simulation (NAS) Program. There were two major reasons for our extensions to UNIX accounting. One, we needed a project accounting system and, two, we had to incorporate numerous extensions to UNIX developed by CRI and by the NAS program. The following paragraphs describe the aspects of the NAS environment and of UNICOS which were the source of requirements for our accounting system.

The Numerical Aerodynamic Simulation Program

NAS is a national facility at NASA's Ames Research Center, Moffett Field, California that began regular operations in July, 1986. It provides computational services to over 650 government, industrial, and university clients at 70 locations throughout the US. The 1987 configuration is shown in Figure 1. The 4-processor 256-Megaword Cray-2 is the major computing resource - the object of these accounting efforts. The Amdahl is used for the Mass Storage Subsystem (MSS) and the Support Processing Subsystem (SPS). These mainframes are linked with communications to local and remote VAXs and workstations. All NAS processors run versions of UNIX.

The Cray-2 provides an effective sustained 80 MFLOPS (Million Floating-point Operations Per Second) on each of its four processors. Aerodynamic researchers apply these computational resources to the complexities of hypersonic and turbulent flow using millions of points in their 3-D computational grids. NAS routinely satisfies requests for as much as 4 hours of Cray-2 processor time on a single problem segment. That amounts to a request for

1,152,000,000,000 floating point calculations, which used to take days. Since most aerodynamic studies require numerous such computational runs, reducing these turn-around times means substantial reductions in the overall time needed to complete analyses. Researchers on NAS have already provided major contributions to aerodynamics and other disciplines.

To gain access to NAS, researchers submit proposals detailing methods, resource requirements, and import of the study [1]. For our current operational year the peer review boards received far more excellent research project proposals than we had resources to accommodate; accordingly, only a fraction of the requested CPU hours were granted to the projects. This heavy demand placed the accounting system in a key project management role.

The Emergence of UNICOS for Cray Computers

NAS provides interactive capability for interpreting supercomputer results on graphic workstations. It is rare that the answer to aerodynamic problems boil down to a few parameters. Rather, there are pressures and velocity vectors over the 3-D space surrounding a 3-D object. Color graphics are the most effective way to display and interpret the millions of data points available. Our most effective graphic workstation programs work interactively with the Cray-2.

The early NAS designers saw that such distributed processing would be optimized with a consistent operating system across all NAS computers. CRI was influenced to implement UNIX on their new Cray-2; NAS took delivery of the second Cray-2 produced in November 1985. The on-site CRI personnel supported getting the new UNICOS operating system running. UNICOS is such a success that CRI has announced

that it will place all further operating system development efforts exclusively into UNICOS; the COS design will be frozen as is.

The Network Queuing System (NQS)

The UNIX interactive environment is not suited to manage as many requests for huge amounts of processing time and memory as was anticipated at NAS; consequently, NAS developed the batch job manager, NQS. As its name implies, the Network Queuing System provides more than just local-host job submission. It extends the batch concept across networked UNIX computers for remote job entry [2]. The location of the job submission has little effect on accounting; remotely submitted jobs must run on a local account.

The typical batch environment requires all processing to be done within jobs. The job submission (in the old days it was the job card deck) has to include not only the processing directives, but also, the account designator, the job priority, and the resources (memory, tapes, CPU time, etc.) required. The job priority and resource requests give the operating system the information it needs for scheduling decisions. Submitted jobs are queued until the required resources are available. Low priority jobs are sometimes held in a queued state for many hours, if not days.

NAS's NQS has been such a successful extension to UNICOS, that it is now distributed and supported by CRI. Since approximately 90% of NAS Cray-2 time is spent running NQS jobs, the functions of accounting and usage controls has been extended at NAS to incorporate NQS queue identifiers, as shall be explained.

UNIX SYSTEM V ACCOUNTING BASIS

Initially, UNICOS (version 1.0) contained just the basic UNIX System V accounting procedures, shown in the simplified data flow diagram of Figure 2 [3].

The *pacct* file accumulates a new record of information whenever a process terminates. This record contains the time of day the process began running, the number of wall-clock seconds, the number of CPU seconds, the number of bytes transferred, the number of I/O requests, and the amount of memory used. Each is identified with the command name, the group ID (gid) and the user ID (uid).

Every night at 4 AM, two *tacct* summaries of this data are produced by *acctprc*. One contains records of totals for each command name and the other contains records of totals for each login name. The command-name totals are system analysis tools, showing the frequency with which commands are used and the average figures for memory use and CPU time. The login-name summaries can be used to keep track of use by individuals. Additional standard utilities, such as *acctdsk*, *acctmerg*, and *prtacct*, provide inclusion of disk use, merging of the daily summaries

into monthly summaries and application of a charging algorithm.

NAS ACCOUNTING REQUIREMENTS

Providing computer time in grants makes our account administration system requirements significantly different from those of centers charging dollars. With direct dollar charges, other centers make general resource availability to users, let them bid for usage (dollar rate being a function of specified priority), and just total up their bill every month. NAS, on the other hand, makes grants of fixed allotments (quotas) of CPU time and disk space to each project. Then, the researchers attempt to complete their research within their established quotas. This is referred to as a "charge-back" system.

Faced with being cut-off after using the allotted CPU time, the research projects have a greater than usual need to monitor themselves. Furthermore, NASA managers who set quotas have to monitor the projects to determine the feasibility of making quota adjustments. Therefore, good tools to monitor account use vs. quota are a must.

The monitoring tools developed at NAS were required to report primarily on a per project basis but there was also a need to provide breakdowns of the individuals' usage of the system. This is complicated by the cases of scientists who work on multiple projects.

ACCOUNTING EXTENSIONS DEVELOPED BY NAS

The *NAS Accounting* system began with the *Job Accounting* software from AIM Technology. This section describes the many modifications we have made to that product over the last year and a half in evolving *NAS Accounting*.

Adding Project Accounting to AIM

As soon as we had completed modifying AIM *Job Accounting* to run on UNICOS, we implemented the needed project accounting. A logical_uid was established for every installed uname-gname pair. For instance, my login name is *eaton*; there are logical_uids for every group I have been a member of, i.e., *eaton.ge*, *eaton.sys*, *eaton.ops*. The numerical logical_uids are assigned sequentially by the utility used to maintain groups and are transparent to the users.

In our system we equated projects with groups. We have approximately 250 projects and assign them names which are simply their gid preceded by the letter g, e.g., *g400*. The users use the *newgrp* command to change accounts. *Newgrp* was a regular user command in UNICOS 1.0 and UNICOS 2.0. When UNICOS 3.0 introduced 4.3BSD-type multi-groups, *newgrp* had to be added as a minor local kernel mod.

In the initial implementation, the *pacct* uids and gids were converted to the uname.gname format and

the logical_uid table was scanned for a match. As we developed over 1000 such accounts, this search began taking an excessive amount of CPU time so a hash function was developed to provide nearly direct access to the logical_uid from the uid-gid pair.

While we developed and implemented this system successfully in 1986, we were not aware that CRI was to unveil a way to do project accounting in UNICOS 2.0. With that release, an "accounting ID" field was added to the *pacct* file to provide a way for users to identify their processing activities as being part of a project. Accounting IDs are maintained in the system in a manner parallel to group IDs. They are mapped to account names. Each user has a set of accounts that he can charge to. He shifts accounts by entering *newacct proj2*. If *proj2* is an account containing the invoking user, the command is successful, and all subsequent processing is tagged with *proj2*'s account ID. The process accounting summary program, *acctprc*, has new options that allow use to be summarized by account names instead of by user name.

Though the Cray extra field allows more user configuration flexibility, an added administrative burden is incurred to establish and maintain both groups and accounts. NAS finds that making projects and groups be equivalent works well. People on projects are sharing a CPU time grant which is logically parallel with the need to share files.

Keeping the base for NAS accounting on standard UNIX user and group IDs allows us to more readily port the accounting into non-UNICOS machines. Though we have UNICOS-unique aspects to our accounting, they are mainly associated with the method for scanning, and the parameters in, the *pacct* file. Those methods and parameter definitions can be readily changed. What would be most disruptive to us would be to have to develop different techniques for administering projects on other supercomputers. NAS is pursuing the acquisition of additional supercomputers and strives to maintain vendor independence. Our implementation allows us to extend our accounting and administrative processes into any UNIX system that will support some version of *newgrp*.

Other Changes to AIM Accounting

Though the extension of AIM to handle project accounting was one of our most significant modifications to the *Job Accounting* product, there were still many aspects that did not suit the needs at NAS. So many changes have been made that there are few unmodified code sections left. The largest unmodified section is the AIM disk accounting. For disk space control we rely on the disk limit system developed here by CRI, which has some sophistication unsupported in the AIM product.

Rate Groups

The AIM job accounting had a rate group concept where every user was a member of a rate group; each rate group could have different charging rates, for different hours of the day and for different days of the week. This was abandoned at NAS in favor of uniform billing rates and UNICOS prime/nonprime routines. The UNICOS routines improved the process time allocations because they divided up long-running processes' time between prime and nonprime time, and also counted holidays as nonprime time.

System Billing Units

Since we do not have any dollar charges, we replaced the dollar designators that appeared in the AIM reports with System Billing Units (SBUs). Regular processing is done at 1 SBU per CPU-minute (0.0167 SBU per CPU second.)

Supergroups

We added the concept of a supergroup to help our center management. All user groups are members of supergroups. The primary supergroups are the NASA research centers at Ames, Langley, and Lewis, the Department of Defense, Universities, Program Support, and Overhead. The amounts of Cray-2 CPU utilization of these supergroups are shown in Figure 3. Program Support and Overhead are for NAS's own computer scientists and system administrators.

The research supergroups are administered by resource monitors, who play a key role in establishing and maintaining all their member projects' quotas. They can query all member groups. They are not able to query outside of their supergroup. For security reasons and to help minimize interproject rivalries, the researchers are only allowed to query on their own group.

Accounting Query Program, *ja*

The AIM accounting query program, *ja*, provided us with an excellent starting point but we have extended it in numerous ways. We still use AIM's natural-English syntax [4] query parser and date conversion routines. All queries have a verb (*get*, *put*), a domain (*my*, *everyone*, *gname*, *uname*, *uname.gname*) and an object (*bill*, *CPU use*, *disk use*). *Get* specifies standard output; *put* directs output to a specified file. The *summary* modifier specifies suppression of the user-level reports, giving just group summaries. The *daily* modifier specifies inclusion of daily information instead of just totals. *For*, *from*, and *to* are for specifying dates or date ranges to override the default - from the first of the month until now.

AIM's responses were verbose; some reports ran on for hundreds of pages. We started piping this output through *pg*, because it was scrolling off our screens too rapidly. Then, we added the ability to request succinct responses with the *use* object, as shown in Figure 4. Many variations on this format provide neat, compact reports in queries for any domain and with any modifier. Some reports include multilevel

totals. When the gname specified is a supergroup, all subgroups are included in the report. These report formats make it easy to scan columns to find significant activity amongst groups, users, or days.

As *ja* became more and more utilized, we tackled the problem of a long initiation time. It took almost one second of Cray-2 time, many seconds of real time, before the *ja* prompt would appear. It was determined that most of this initiation time was spent reading in and dynamically allocating storage structures for the *ugroup* file text data which contains the set-up parameters for all the groups. A program was created to transform *ugroup* into an include file, which was then compiled into *ja*. Now, initiation time is about one tenth of what it was.

Inclusion of New Measures of Resource Use

Where AIM's *ja* code handled four compressed CPU time measures per account-day, NAS decided to extend this to 18 floating-point numbers to keep records of other measures of resource use. We also

wanted to capture some of the data from a CRI accounting extension (UNICOS 2.0) for multitasking (multiprocessing). The multitasking breakdown are appended, if needed, as an additional record in the *pacct* file. These measures can be retrieved as shown in Figure 5.

Subsequent analysis of this data provides NAS with measures of how different groups of users load the system. It also provides us with basis data for consideration of implementing a more complex charging algorithm.

NQS's Deferred Queue Accounting

Special provisions have been built into our accounting to recognize processes from jobs that have run in NQS's deferred queue. The deferred queue is for jobs that will be run only during periods when there is excess CPU time available [5]. In Figure 5 note that processing times for deferred queue jobs (rows 5 and 6) have a zero SBU charge rate. NAS Accounting recognizes processes' NQS queue from

```
NASacct 3.3->get eaton use
```

	1KB blks	MINUTES	SBU	GROUP QUOTA	USED
>From 12/01/87 to 12/08/87					
eaton.sys	141	2654.9	1406.9	10800	13.0%
eaton.ops	5798	21.8	21.8	2820	0.7%
total for eaton.	-	2676.7	1428.7	-	-

Figure 4

```
NASacct 3.3->get eaton.ops cpu use
Statement for eaton.ops.
```

```
eaton.ops CPU use from Monday, December 1 1987 at 00:00
through Sunday, December 6 1987 at 23:59
```

Type of use	Amount	Rate	SBUs
prime interactive cpu	252.122847	0.017	4.20
nonprime interactive	107.844979	0.017	1.80
prime batch cpu secs	406.320387	0.017	6.77
nonprime batch cpu	0.000000	0.017	0.00
prime deferred queue	0.000000	0.000	0.00
nonprime deferred que	206.140877	0.000	0.00
prime kword-minutes	596.200750	0.000	0.00
nonprime kword-mins	234.403083	0.000	0.00
prime bytes xferred	730250295.000000	0.000	0.00
nonprime bytes xfrrd	711152825.000000	0.000	0.00
prime physcl I/O reqs	61000.000000	0.000	0.00
nonprime phs I/O reqs	10022.000000	0.000	0.00
count of processes	5571.000000	0.000	0.00
run-time sbu	0.000000	0.000	0.00
user time w/ 1 proc	624.565625	0.000	0.00
user time w/ 2 procs	0.000000	0.000	0.00
user time w/ 3 procs	0.000000	0.000	0.00
user time w/ 4 procs	0.000000	0.000	0.00
Total:			12.77 SBU

Figure 5

codes we have placed in UNICOS's *Account ID* field.

The deferred queue helps NAS accomplish two goals. First, it allows users to finish up their work when they are nearing their CPU quota. Before we had deferred queue, many users worried so much about running out of CPU time that they used almost none while some went for broke and hoped they would get their quota increased. Second, it provides an encouragement for users to submit jobs for weekend runs. Before the deferred queue was implemented, the Cray-2 tended to complete all submitted jobs during the weekends and sit idle. With the deferred queue, we now have more jobs queued up to run in its low-priority, no-charge mode.

NAS now has a simple charging algorithm largely because the researchers negotiated for a certain amount of CPU hours. It is not unreasonable to ignore the other usage factors, as the Cray-2 almost never has to wait for I/O or memory.

Reports

Accounting reports are especially important in providing timely and accurate information on system use and availability to all levels of NASA management. Reports also provide insights needed to make

system configuration decisions.

Supergroup and Project Use Summaries

Figure 6 shows the result of feeding everyone's *ja* data into procedures of *awk*, *ingres*, *c*, *tbl* and *xroff* to provide a top-level view. This is one of many NAS-accounting-based summaries produced weekly and monthly. The most detailed one provides, for each supergroup, a table with rows for each project that contain:

- the project name and, if not NASA, the project sponsor
- CPU minutes of major users
- CPU minutes of last week/month
- CPU minutes total to date
- CPU minutes of deferred queue to date
- System Billing Units total-to-date
- quota
- SBUs left

<i>Class</i>		Interact. <i>minutes</i>	Batch <i>minutes</i>	dfrd-q <i>minutes</i>	cpu <i>total</i>	ave size <i>core MW</i>	I/O <i>MWords</i>	<i>k reqs</i>
NASA_ARC	prime	448	2539	908	3895	6.694	8977	8627
	nonprm	159	2752	4435	7346	17.399	6247	881
NASA_LaRC	prime	509	3835	538	4882	5.332	2319	2025
	nonprm	246	4178	2643	7067	6.931	3380	1071
NASA_LeRC	prime	8	705	23	735	34.538	263	38
	nonprm	0	242	67	309	32.733	71	9
Other_NASA_	prime	43	115	0	158	6.702	132	69
	nonprm	4	110	0	114	9.780	40	13
DOD	prime	66	1246	281	1593	13.884	3032	331
	nonprm	25	1297	1829	3151	6.547	1151	187
Commercial	prime	25	529	0	554	14.160	342	125
	nonprm	18	146	0	163	14.108	44	26
University	prime	5	2	538	544	8.340	140	27
	nonprm	12	738	1446	2196	16.267	533	72
PRGSUP	prime	154	239	0	393	6.463	1326	3108
	nonprm	65	137	0	202	2.277	798	1574
OVERHEAD	prime	158	18	0	176	0.072	2458	10532
	nonprm	560	1	0	560	0.282	11126	12545
<i>total</i>	<i>prime</i>	<i>1416</i>	<i>9227</i>	<i>2288</i>	<i>12931</i>	<i>8.944</i>	<i>18988</i>	<i>24883</i>
	<i>nonprm</i>	<i>1088</i>	<i>9599</i>	<i>10420</i>	<i>21108</i>	<i>11.718</i>	<i>23391</i>	<i>16377</i>
	<i>total</i>	<i>2505</i>	<i>18827</i>	<i>12709</i>	<i>34040</i>	<i>10.664</i>	<i>42380</i>	<i>41260</i>

Figure 6

Time-of-Week Summaries

The *pacct* accounting file has a wealth of data that we are exploiting to display patterns of use over the week. An example is shown in Figure 7. The pattern of use shifting between interactive and batch is clearly shown, as are distinct times when the system was out of service. These changing system load characteristics help in setting up optimal UNICOS and NQS configurations. Some NQS parameters are automatically adjusted at certain times of the day and week to take advantage of known usage patterns.

Often such charts are made with sampled data, but ours portray the actual use during each 20-minute interval. To portray CPU use distribution, such as in this chart, it is not enough to process the *pacct* records. The process table is scanned every twenty minutes, summarizing the active processes' accumulated CPU time so that the CPU time for lengthy processes is correctly distributed.

Automatic Account Controls

Our extensions to accounting provide direct disabling of regular processing activities when a project's CPU quota is exhausted. Nightly, *NAS Accounting* checks all user groups' use against their quota. If a group's CPU use reaches 90% of its CPU quota, the group's members are warned via electronic mail. If a group reaches 100% of its quota, the members are informed that they have severe limits on CPU time and memory use of processes run outside of the deferred queue. A UNICOS extension to UNIX permits setting such process limits [6]; a local kernel mod adjusts the settings as a function of the group to be charged. The kernel mod includes logic to recognize NQS processes. NQS normally adjusts process limits to be greater than those for interactive processing.

This automation saves us many hours of otherwise manual efforts. As a side-benefit, the *jaq* command was added that provides a quicker method than *ja* for determining account status.

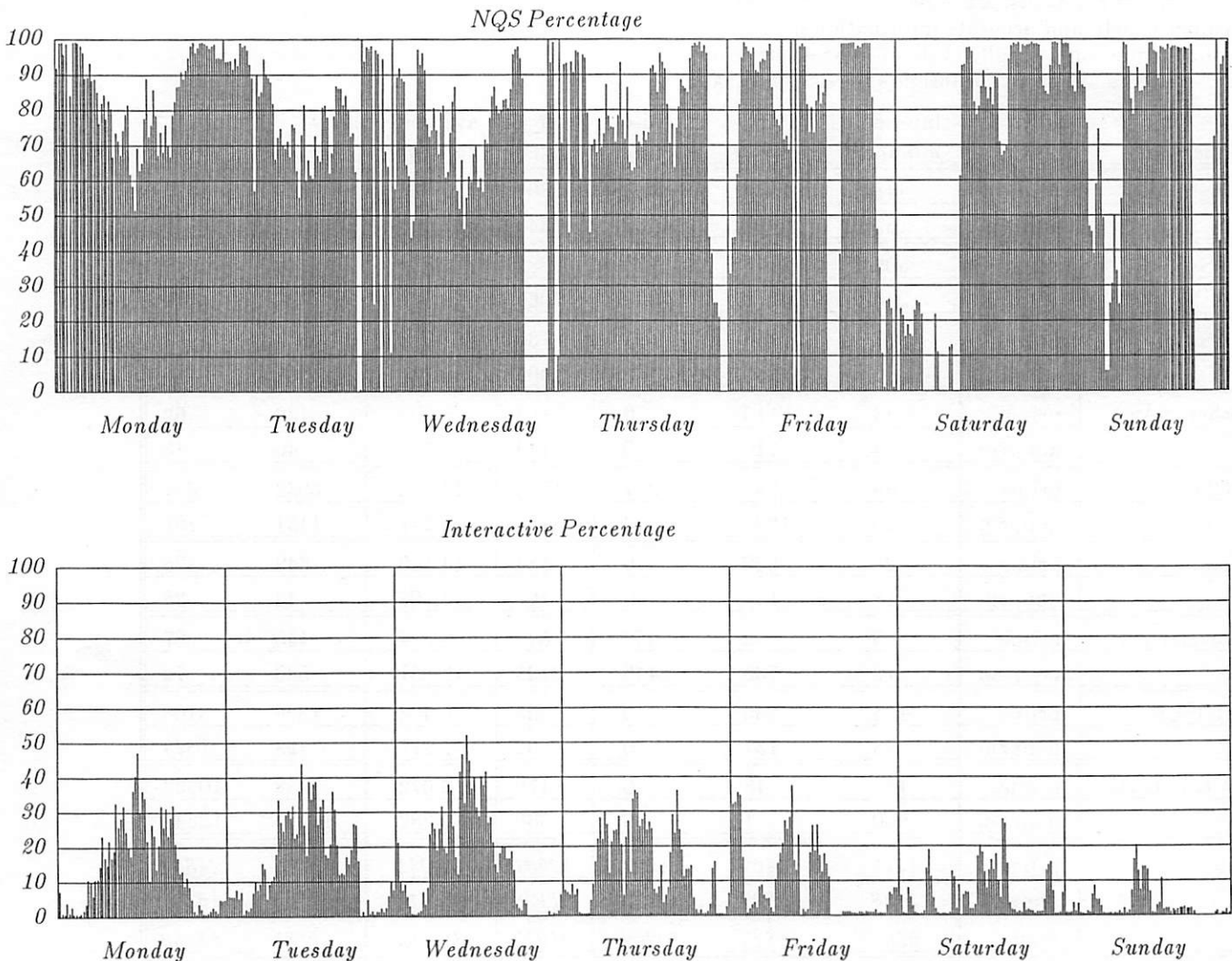


Figure 7

Plans

We are planning to move more of the statistics and data off the supercomputer(s) and into a support processor data base. Such a data base will off load the supercomputer, merge multi-system accounting, and provide more efficient and flexible report generation.

DISK ACCOUNTING

Group disk use is tracked and limited by a UNICOS extension developed at NAS by CRI. The limit system tracks each groups' disk use on each file system by making adjustments whenever disk blocks are allocated or freed. Write errors occur when the group's limit on disk blocks on a file system are exceeded.

Each research group has access to three different types of file systems - permanent (home) store, scratch, and /tmp. The scratch file systems are intended for large active and/or short-term files. Nightly, files that have not been accessed or modified for over 3 days are removed.

There is no effort being made to charge SBUs for daily disk use so the AIM *Job Accounting* code for disk accounting is not needed at NAS.

CONCLUSION

We have evolved a unique accounting system far different from the AIM *Job Accounting* we started with. Some aspects of the AIM *Job Accounting* were not germane to our center and were deleted. Needed capabilities have been added that provide supergroups, succinct reports, and extended resource use characterization.

Though CRI has established a new field in their process accounting to handle project accounting, NAS has used the group identifiers for this purpose. The NAS method has administrative simplicity and it can be ported to other machines more readily.

Additional utilities provide a variety of tabular and graphics reports. Group CPU and disk use are automatically tracked and shut down when quotas or limits are exceeded.

The NAS algorithm for calculating SBUs is simple; it corresponds to the resource estimates of projects requesting grants. Incorporating other usage factors into our SBU calculations is inhibited by the added complexity it would bring to the arrangements made with these researchers.

The NAS project accounting system is just one component in what NASA intends will be a center that provides not just the best computational resources for pioneering scientific research, but a center that will also be a pathfinder for the supercomputing community. We at the program are fortunate to have such an opportunity to contribute to computer science generally and to UNIX in particular.

ACKNOWLEDGEMENTS

Acknowledgements are due to all those people who have contributed to the operational systems at NAS. Special recognition is extended to Doug Anderson for AIM *Job Accounting* mods, John Musch for UNICOS mods, Murali Sundaresan for NQS mods, Bill Kramer, NASA's Cray-2 Manager, for guidance and support, John Pew for *pic* chart development, and Emma Eljas for editorial support.

REFERENCES

- [1] Bailey, F. Ron: "NAS - Current Status and Future Plans", *Supercomputing in Aerospace*, NASA Conference Pub. 2454, Mar. 87.
- [2] Kingsbury, B. K.: *The Network Queueing System*, NASA Contractor Report 177433, December 1986.
- [3] *UNICOS Administrator Commands Reference Manual*, SR-2022, Cray Research, Inc., 1986.
- [4] *AIM Job Accounting, Release 1.0, User's Manual*, AIM Technology, 1986.
- [5] Kramer, W. T. C: "System Wide Performance Measurement for the Cray-2", *Cray Users' Group Proceedings*, Fall 1987.
- [6] *UNICOS System Calls Reference Manual*, SR-2012, Cray Research, Inc., 1986.

The first of these is the fact that the
the second is the fact that the
the third is the fact that the

THE SECOND OF THESE

The second of these is the fact that the
the third is the fact that the
the fourth is the fact that the

The third of these is the fact that the
the fourth is the fact that the
the fifth is the fact that the

THE THIRD OF THESE

The third of these is the fact that the
the fourth is the fact that the
the fifth is the fact that the

The fourth of these is the fact that the
the fifth is the fact that the
the sixth is the fact that the

The fifth of these is the fact that the
the sixth is the fact that the
the seventh is the fact that the

The sixth of these is the fact that the
the seventh is the fact that the
the eighth is the fact that the

The seventh of these is the fact that the
the eighth is the fact that the
the ninth is the fact that the

The eighth of these is the fact that the
the ninth is the fact that the
the tenth is the fact that the



Scott D. Carson
Department of Computer Science
University of Maryland
College Park, MD 20742

Using Groups Effectively In Berkeley Unix

ABSTRACT

In the Berkeley version of the UNIX operating system, files have an associated group-id and users each have an associated set of group-ids. Whenever a file's group-id is a member of a user's set of group-ids, a specific set of permission bits is used to determine access to the file. While the group mechanism is intended to provide an easy way for users to share files, in practice a number of pitfalls arise that prevent users from sharing files in a convenient manner. This paper identifies the pitfalls inherent in the current implementation of the group mechanism, and proposes a modification to the mechanism that eliminates them.

Introduction

A *group*, in the UNIX context, is a collection of users who need to share access to a set of files. In some versions of UNIX, a particular group-id is associated with each process and each file; if the two match, then the process's access to the file can be determined by the file's set of "group protection" bits. As of 4.2 BSD, Berkeley extended this concept to allow a process to belong to a *set* of groups simultaneously [1]. The intent of the extension was to allow users to work in groups more effectively by eliminating the need to switch their processes between groups.

Unfortunately, a number of years later, this concept remains relatively unexploited. The primary reason is that, while "group write" permission is usually required for files that are to be shared, users do not generally allow "group write" permission by default. Further, if users did allow "group write" by default, they would expose their "private" files to writing by other group members. The result is that, even though users try to work in groups, they tend to create files that are inaccessible to other group members.

This paper describes a technique for making the group concept work effectively, providing and preserving write access for group members when desired, and providing write-privacy when appropriate.

The Problem

When a file is created, it is assigned an owner, a group, and a set of protection bits. The owner of the file is determined by the user-id of the creating process, and the group is inherited from the directory in which the file is created. The protection bits, also known as the *file mode*, specify read, write, and execute permission for three classes of users: the owner, members of the file's group, and other users. These bits are typically encoded as an (octal) number whose

digits represent the three bits, in order, for each of the three classes. For example, mode 0644 allows read and write access for the file's owner, but only read access for group members and others.

By default, a file's protection bits are determined by the *umask* of the creating process. More specifically, a file's protection bits are the intersection of the set of permissions specified in the creating system call (*open(2)*, *creat(2)*, or *mkdir(2)* [2]) and the inverse of the *umask*. For example, setting the *umask* to the value 022 causes newly created files to be (typically) of mode 0644, while setting the *umask* to zero causes newly created files to be of mode 0666.

Groups are collections of users who share files, often while working on a common project. A typical scenario for file-sharing is as follows. A directory is created, and its group is set to the appropriate group-id. The mode of the directory is typically 0775, which allows members of the group to create files, delete files, and search the directory. The group members then manipulate files in the directory as required. These files, when created, inherit their group-ids from the directory, and ideally, members of the group can all access the files with equal facility.

In practice, however, such a scenario often does not work. The fundamental reason for the failure of the scheme is that users typically have their *umasks* set to 022, which prevents writing by group members. Thus, when users create files that are to be shared, they prevent other members of the group from writing the files. Even if the shared files are explicitly set to mode 0664, which allows group write access, the mode can be altered, for example, by editors that create a new copy of a file after modification. The result is that whenever a user modifies a shared file, he or she must first make a copy of the file (which is owned by the modifying user) and delete the original. This procedure, while functional, is both awkward

and error-prone.

The solution to this problem is for users to set their umasks to the value 02, which allows group writing by default. Unfortunately, this introduces another problem. The home directory of each user has an associated group, as do all of each user's private files. Setting the umask to 02 by default, then, would allow unwanted write access to users' files. This solution, while making file-sharing work well, makes it difficult to maintain files that are private.

In fact, the secondary problem described above is exacerbated by a common misuse of groups by UNIX system administrators. Since every file must have *some* associated group, there is a tendency to create groups that exist for the purpose of classifying users rather than sharing files. It is common to see groups such as "staff", "faculty", and so forth that exist solely because the system administrator could not think of any better group-id to use when creating a new user's home directory.

Solution Criteria

The obvious criteria for the success of a scheme to solve the problem described above are that the scheme must provide the ability to share files when sharing is desired, and that the scheme must provide the ability to avoid sharing files that are to be private. Further, considering that users often work in multiple environments simultaneously, the scheme must not require that the user type commands that change the umask whenever a transition from work on shared files to work on private files is made. These properties can be stated as follows:

1. When a file is made sharable, it stays that way without further action.
2. When a file is made private, it stays that way without further action. Further, all files are made private by default.
3. No action is required when changing between environments in which files are to be shared and those in which files are to be private.
4. The distinction between private and sharable files must remain irrespective of copying, editing, or other action.

In addition, a solution should not require extensive modification of software; it must be easy to implement.

The Solution

The solution to the problem of using groups effectively can be derived by reconsidering the roles of group-ids and files. Rather than requiring that each file have an associated group, the association between a group-id and a file can be made optional, to be used only when the group effect is actually desired. In other words, files that are to be shared have an associated group-id, while those that are to remain private

have no associated group-id.

This can be implemented by creating a distinguished group, called the "none" group, of which no user is a member, and by having each user set his or her umask to the value 02. Private files, including each user's home directory, can be owned by the "none" group; since no user is a member of the "none" group, the fact that group write permission is allowed by default is inconsequential. The umask value 02 does allow group write access on files that belong to groups other than the "none" group.

This solution introduces another problem. The program that allows users to change the group-id of a file, called *chgrp*(1) [2], only allows users (other than the superuser) to set the group-id to that of a group of which the user is a member. Since no user is a member of the "none" group, *chgrp* does not allow any user to associate the "none" group with a file. Fortunately, there is a simple solution to this problem: make *chgrp* treat the "none" group differently, so that any user can associate the "none" group with a file. The test in *chgrp* that determines whether or not the user can set the group-id of a file to a particular group normally reads

```
if (superuser ||
    user belongs to target group)
    allow user to change group-id.
```

After modification, the test reads

```
if (superuser ||
    user belongs to target group ||
    group is "none")
    allow user to change group-id.
```

In order to implement this solution, two modifications to the system must be made. First, *chgrp* must be modified as described above. Second, the "none" group must be created by adding a line to the file */etc/group*. Additionally, the home directories of users must be associated with the "none" group when they are created, each user's umask must be set to 02, and care must be taken to ensure that no user is ever made a member of the "none" group.

Solution Evaluation

For this solution to be effective, it must meet the criteria described in Section 3. In addition, it must be shown that the modification to *chgrp* does not create additional problems.

First, consider the criteria established in Section 3. The first requirement is that shared files remain sharable. Suppose that a user creates a directory in which shared files are to be placed, and associates a specific group with that directory. Whenever group members create files (including directories) in the shared directory, those files inherit the directory's group-id, and since each user's umask is set to 02, all files created in the shared directory remain group-writable. Thus, the first criterion is satisfied as long as the shared files remain in a shared directory.

The second requirement is that each user's private files remain private, and that files are private by default. Suppose that, when a user is added to the system, that user's home directory is set to belong to the "none" group, and that the user's default umask is set to 02. Any files (including directories) that are created in the user's home directory are set to allow group writing, but since each file also inherits the "none" group, of which no user is a member, the group write permission has no effect. If a user never tries to share files, for example, using the *chgrp* command, then all of the user's files that are created in the home directory subtree belong to the "none" group. Thus, without action to the contrary, files in the home directory subtree are private when created and remain that way. In addition, any files that are created in a private directory are themselves private by default.

Some care is required to retain privacy when files are created in places other than the home directory subtree, since newly created files inherit the group-id of the directory in which they are created. In practice, only a few directories, such as */tmp*, are used as common repositories, and these should naturally be owned by the "none" group.

The third requirement is clearly met, since a user's umask need never change in this scheme. The fourth requirement, that no action other than an explicit one change the distinction between shared and private files, is met as long as files are not moved between shared and private directories.

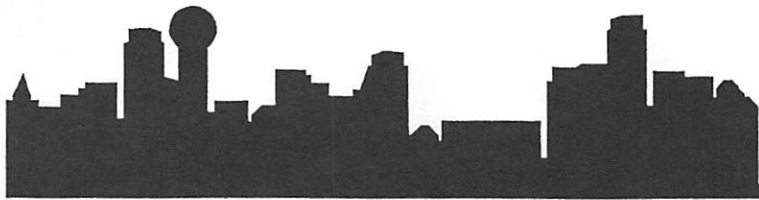
Now, consider the modification to *chgrp*. All criteria used to determine whether or not a user can change the group-id of a file remain in place for groups other than the "none" group. In other words, for groups other than the "none" group, the effect of *chgrp* remains unchanged. For the "none" group, all existing criteria remain unchanged except one: that the user belong to the target group. Therefore, modifying *chgrp* produces only the desired effect, without introducing additional problems.

Conclusion

This paper has identified the fundamental problem that prevents the effective use of the UNIX group mechanism: that there is no way to associate *no group* with a file. Criteria for a correct solution have been identified, and a solution that meets these criteria has been proposed. The solution is simple to implement, requiring minor changes in one code module along with a trivial change in administrative procedures. The result of applying this solution is that users are able to exploit the utility of the group concept.

References

- [1] Leffler, Samuel J., "Bug fixes and changes in 4.2BSD," CSRG, Berkeley, July 28, 1983.
- [2] *Unix Programmer's Manual*, Volume 1.



G. Winfield Treese
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
treese@ATHENA.MIT.EDU

Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD

ABSTRACT

4.3BSD UNIX as shipped is designed for use on individually-managed, networked timesharing systems. A large network of individual workstations and server machines, all managed centrally, has many important differences from such a model. This paper discusses some of the changes necessary for 4.3 in this new world, including the file system layout, configuration files, and software. The integration with Athena's authentication system, name service, and service management system is also discussed.

Overview

"By 1988, create a new educational computing environment at M.I.T. built around high-performance graphics workstations, high-speed networking, and servers of various types." This one-sentence statement is a high-level description of the technical goals of Project Athena. While the primary goals are to enhance education, attaining them has required a significant effort to engineer a software system for use in such an environment.

The Athena hardware environment currently consists of approximately 650 workstations and 65 dedicated server machines. There are two kinds of workstations: DEC MicroVAX systems and IBM RT PC's. The servers are VAX 11/750's or dedicated workstations of either type. The operating system in use now is 4.3BSD UNIX on the VAX machines, and IBM's 4.3/RT UNIX for the RT PC systems. All systems include support for Sun Microsystem's Network File System (NFS).¹ The workstations and servers are connected to local-area Ethernet subnetworks, which are linked by a high-speed fiber optic "spine." At present, there are twelve such subnetworks at M.I.T.

The problems of a distributed system are the scale of the operation and the role of the network as a fundamental component. UNIX systems have traditionally been managed on a "one system, one wizard" basis, but this is not acceptable at an eventual scale of 1000 workstations, 100 server machines, and 10,000 users. Two questions often asked are: "Does it scale?" and "Is it well-behaved on the network?" All too often, the answer to one or the other is "No," and part of the system must be reworked to satisfy those constraints.

This paper describes the goals and constraints faced by Athena, as well as many of the solutions devised in building such a system. In particular, the next two sections examine the goals and evolution of the

computing system side of the Project. Next is a discussion of the base operating systems in use, such as 4.3BSD. This is followed by a description of the constraints of scale and the network, as well as some of the solutions bounded by those constraints. Finally, system configuration issues and future directions for enlarging the workstation base are discussed.

Goals

The following are four important goals for engineering a Project Athena workstations.

The system must provide a coherent environment across heterogeneous hardware. When Project Athena was created, "coherence" was identified as an important characteristic of a successful workstation environment. Briefly, "coherence" means that the environment seen by users on different workstations should be as similar as possible. To a first approximation, this is achieved by using Berkeley UNIX systems on all workstations.

It must provide rich computing environment for the Athena user community. To make a workstation optimal for educational use at M.I.T., a rich software environment must be available. If, for example, the software is not useful for building tools for teaching a thermodynamics class, that class will not use Athena workstations. Many third-party software packages, including an editor, text formatter, and spreadsheet, have been added to the standard UNIX system for this reason.

It must scale to 1000 machines such that the Athena Operations staff can manage them. The entire network of workstations must also be manageable. The Athena Operations staff is not large, but it is currently responsible for over 700 machines. Hence, a solution to a problem that requires any resources in proportion to the number of machines (e.g., having to visit each workstation) is

prohibitively expensive. Wherever possible, operations tasks should be automated; otherwise, they should at least be performed centrally.

It must behave well on the network. Athena workstations must also be "good neighbors" on the network; they should not generate problems for other systems on the network, such as by gratuitously consuming network bandwidth.

Evolution of Project Athena

At the time of Project Athena's inception in 1983, workstations as defined here were still in the design phase. In the beginning, then, Athena used off-the-shelf hardware and software. Approximately 50 DEC VAX 11/750 systems were deployed as traditional timesharing systems running Berkeley UNIX (4.2BSD, later 4.3), and over 100 IBM PC/AT's running PC-DOS were deployed as networked single-user machines. Using both systems yielded much information about managing and engineering networked UNIX systems and single-user machines.

Workstations became available to Athena staff in late 1985. At that time, Athena attempted to apply the lessons learned from running timesharing systems to distributed workstation systems, as well as to develop solutions to the new problems posed by workstations. Much of the design work was done "on the fly" in order to make usable workstations available to staff, and eventually to users, as quickly as possible.

The first student workstation clusters, running a prototype system, were opened for use in the fall of 1986. Since then, the number of workstations available has steadily increased. During the 1986-87 academic year, both timesharing and workstation systems were in use. Over the summer of 1987, the Project shifted almost entirely to the workstation environment, with only a few timesharing systems in use for classes with specific requirements (e.g., those using a non-networked database system). The majority of the timesharing systems were converted to NFS file servers, primarily for student "lockers," or file storage areas.

Hardware

The typical Athena workstation is roughly a "3M" machine; that is, it has a 1 million-instructions-per-second processor, a megapixel (1000 x 1000) display, and three or four megabytes of memory. It also has a mouse, a local disk (typically 30 - 70 megabytes), and an Ethernet network interface. Actual hardware in use includes DEC VAXstation II and VAXstation 2000 systems, and IBM RT PC desktop models. Various other configurations of MicroVAX and RT PC systems are used for development and as dedicated servers.

The heterogeneous hardware base has imposed certain constraints on Athena software. For example, the VAX and the RT PC architectures use different

byte orders to represent integers. Hence, if data is to be exchanged between the two architectures, the software must be prepared to handle this difference. This is true both of network protocols and of files used for data storage, since such files may be transferred between machines. Some software, such as NFS, already obey this constraint; all new software must also obey it.

The hardware differences also make the goal of coherence difficult to reach. For example, the compilers available on the two machines are somewhat different. While the differences are minor, they can be most annoying. Since the same source code is used on both machines whenever possible, the standard source pool has been partitioned into sections of machine-independent and machine-dependent sources to simplify the system build process. The compilers, for example, are machine-dependent; a text editor such as */bin/ed* is not.

Of course, hardware from other vendors meets the specifications for the basic Athena workstation. In the future, the "Athena Environment" will be produced as a layer on top of vendor-supplied systems that meet certain fundamental requirements.

Base Software Systems

The basic software used on all Athena workstations at this time is 4.3BSD UNIX with machine-dependent software supplied by the hardware vendors. All systems include NFS client support and use the X Window System.² The standard Berkeley distribution is augmented by several third-party software packages and local Athena modifications and additions.

RT PC System. The RT PC kernel is taken from 4.3/RT. NFS support has recently been integrated into that kernel at Athena. The necessary machine-dependent utilities (e.g., the C compiler) were also drawn from the 4.3/RT system.

VAX System. The basic VAX system is 4.3+NFS from the University of Wisconsin. Some device drivers from Digital's Ultrix-32 have been added to handle some of the hardware in use at Athena.

Constraints of Coherence.

Providing a coherent environment across different types of workstations imposes several constraints on constructing the system, including the following.

The same basic system must be available on all platforms and must evolve in synchrony. To promote both uniformity and maintainability, all software is periodically built from the source code. This ensures that all changes to header files, libraries, etc., are actually reflected in the running system. Maintaining and building sources for two different architectures turns out to be no small task. Unfortunately, *make* and *rdist* alone are not sufficient to

maintain source code on multiple architectures, especially when some changes (e.g., kernel modifications) must be made to code that is similar but not identical. New tools to automate this procedure are under development.

Data must be interchangeable between the systems. Since different machines may use different internal data formats, applications must be prepared to cope with such variations. This is particularly true of network services, since services may be available from servers with different architectures. Fortunately, this constraint is not difficult to observe, *provided it is anticipated*.

Constraints of Scale

Scale is one of the driving considerations in building the Athena system. Unfortunately, not all constraints of scale are obvious at first glance (or even the second or third). Some of the specific constraints of a large scale system include the following.

Resources cannot be expended in proportion to the number of workstations. This may seem obvious, but it is a constant problem. Athena Operations, for example, does not have resources that grow in proportion to the number of workstations installed. The only way to support those workstations is to make operational tasks more efficient.

Differences in software, including configuration files, must be minimized. One way to minimize operational tasks is to minimize the differences between workstations. There are more than twenty different configuration files in a standard 4.3 system. Workstation systems are much easier to manage when most of these are identical from workstation to workstation. On Athena public workstations, all configuration files are identical except for */etc/rc.conf*, which contains the hostname and network address. Operators can be trained to understand two or three configurations, but expecting them to understand the subtleties of 1000 is too much. If configuration files are identical, an operator can simply look at the configuration file that seems to be incorrect, realize that it is not standard, and copy the correct version from a standard source.

There are, of course, some small exceptions to this rule. In particular, a workstation must be able to find the nameservers in the first place. At this time, a list of nameservers is maintained on the workstation's local disk. In the future, this should not be necessary (see "Future Plans").

Network services must be redundant to tolerate network and server failures. If a central name or authentication service is not available, the world will grind to a halt. Providing redundant servers minimizes the probability of a given service being unavailable, as well as distributing the load over several different servers when all is well. In contrast, it is not always possible, nor is it necessary, to

provide redundant servers for modifying central database information that is later distributed to appropriate servers (e.g., the central nameserver database). As a corollary to this, critical network services should also rebound quickly after system crashes or power failures.

A centrally-managed name service should provide information for finding network services. Suppose the FOO service is provided by machine HERA.MIT.EDU. If this service moves to ZEUS.MIT.EDU, it should not be necessary to update configuration files on 1000 workstations. Indeed, experience has shown that there would be workstations with the wrong information months after the change takes place. A name service provides a much easier way to manage this problem.

Software installation must be quick and painless. Installing new software on a workstation should not take much time, nor should it require many props. Since workstation systems at Athena are installed to a standard form, the installer need perform very little customization; the rest of the procedure is automatic. In fact, for most workstations, only the hostname is different.

Software update must be quick and painless. Automatic update is even better, if it works correctly. In an ideal world, software update would be completely automatic. In an automatic update, a workstation compares its software to a central library and makes any necessary changes. Some software must be updated with great care, however. For example, a new kernel requires that the workstation be rebooted, and often requires that various kernel-dependent utilities, such as *ps*, be updated at the same time. The tools for doing this are not yet entirely reliable.

Configuring servers must be quick, painless, and reproducible. Service machines (e.g., printer servers) are implemented as a set of differences from a basic workstation system. Converting a standard workstation to a server is then a straightforward process. This allows a vanilla machine to be "swapped in" quickly for a server should hardware problems arise. It also simplifies updating servers with the latest software release.

Security must not depend on superuser access. In a large environment such as Athena's, the root password to workstations cannot be different on each machine. Indeed, with the workstation physically available to a user, it is not necessary to know the root password; booting the machine in single-user mode is sufficient to gain root access. A user may therefore easily modify the software local to the workstation, and network services must not blindly trust root users from workstations.

Network services must be managed centrally. Trying to manage a large number of network services can become quite difficult. When only 10 or 100 workstations are involved, it may be possible to

manage services on one or two machines, a job that can be done by hand. On a larger scale, the number of servers must also be considered. To this end, Athena has designed the Service Management System (SMS),³ which consists of a central database of service information, tools for manipulating that information, and tools for extracting appropriate information for the services themselves. A particular advantage of maintaining this database is that it reduces to one the number of different places a given piece of information must be stored: it is kept once in the database and is provided to servers as they need it.

Constraints of the Network

Working in a networked environment also imposes several constraints on the overall system. One of the most important for users is that several services available on timesharing systems should be available on workstations as well. In some cases, these services behave somewhat differently than in the timesharing world, but functionality is preserved. Some of the services include the following.

System Libraries. Timesharing systems have many programs available for a user to execute; most of these are traditionally found in */usr*. Workstations do not typically have the hundreds of megabytes necessary to keep the complete system library available on a local disk, and it would be impossible to keep the software up to date if they did. To solve this problem, Athena has provided a "Remote Virtual Disk" (RVD) service. RVD was originally developed at M.I.T.'s Laboratory for Computer Science and significantly enhanced at Project Athena. An RVD "pack" appears to a workstation just as a local physical disk device does. An RVD server only supplies requested disk blocks; all filesystem information is manipulated by the client workstation. As a result of this, an RVD pack may be used by many clients in a read-only mode, or by a single client in an exclusive read-write mode. Another effect of the block-level service is that a single VAX 11/750 server can support 75 client workstations with quite acceptable performance.

Name Resolution. On a timesharing system, names are often translated to machine-usable data by configuration files. For example, */etc/hosts* maps machine names to numeric Internet addresses. Another file might contain the name of the RVD server that provides the system libraries. In a large, dynamic environment, this reliance on static files causes innumerable problems. To manage changing information, it is necessary to have a service that provides name translation. Athena undertook some minor extensions to the BIND⁴ nameserver package from 4.3BSD to provide generalized name service; the resulting software is known as *Hesiod*.⁵ *Hesiod* provides information on users, locations of user lockers, locations for various network services, etc. Changes in the *Hesiod* database, which is generated by the Service Management System, are available to all

workstations within a few hours (the delay is caused by time to propagate to the Hesiod servers and by internal nameserver caches of the BIND implementation).

Authentication Service. UNIX systems have traditionally stored encrypted passwords in */etc/passwd* on each machine. It is quite difficult to maintain password and group files on 50 timesharing machines and completely impractical to provide complete password and group files for each machine in a large network, so an alternative authentication system is required. Athena has implemented a system known as *Kerberos*⁶ to handle authentication for network services, including workstation login.

File Service. A user's files should be available for use on any workstation. At Athena, each user has a "locker" for personal storage. User lockers are distributed across many file servers, but the appearance to the user is that the home directory is as expected, and it is the current directory at login, just as on a timesharing system. One difference is that other users' home directories are not immediately available but must explicitly be attached to the workstation's directory hierarchy. The net effect, however, is positive: in the earlier days of Athena timesharing, a user could not gain access another's files if they were on a different machine. Security and privacy considerations are observed, however, and the usual UNIX protections are enforced.

Printing. Typical timesharing systems have printers available locally. The environment assumed for 4.3BSD makes some attempt to provide networked printing facilities, but the *lpr* system is still difficult to manage. Two of its major problems are local queuing and local configuration. Normally, *lpr* drops a print request in a queue on the local machine, and a line printer daemon either prints it or queues it to a remote machine, as specified in */etc/printcap*. *lpr* gives the user no indication whether or not the machine physically connected to the printer is actually up and accepting requests. In the Athena workstation world, this can cause a file to be left in a local queue on a workstation, with no guarantee that it will ever be printed. To circumvent this problem, *lpr* has been modified to queue directly to the remote printer server. If it is not available, the user receives an immediate error and may try to find another printer to use.

The second problem is */etc/printcap* itself. As a static configuration file, the copy on every workstation must be updated when a new printer is added or an old one moved. *lpr* has been modified to query *Hesiod* for information about printers in order to find a printer server. Printer servers themselves behave more traditionally; they use a local */etc/printcap* for detailed information about their own printers.

Electronic Mail. Timesharing systems provide a convenient maildrop for users. The address is typically *user@machine*, and local users can be addressed

simply as *user*. Where, then, should mail be kept for a user who might login on workstation ABC one day and workstation XYZ the next, especially when none of the workstations has sufficient local disk space to store mail? Athena has adopted the concept of a "post office," which holds mail for a user until he picks it up. The Post Office Protocol⁷ software supplied with the MH mail system⁸ has been modified for use with *Kerberos*, and the retrieval software finds the mailbox using *Hesiod*. The changes are transparent, so the user simply uses the same commands as he did on the timesharing system, without having to worry about the details. At this time, Athena has three post offices in use, serving approximately 8000 users.

Sending mail to Athena users is also straightforward. The address is simply *user* within Athena, or *user@ATHENA.MIT.EDU* from outside. All mail is routed by a central mail hub, which uses a master list of users at various post offices. The list is provided by SMS. This machine also handles distribution to mailing lists within Athena. When a user sends mail from a workstation, it is queued to the mail hub for forwarding to a post office, mailing list, or outside machine, as appropriate. Note that *all* mail goes to the mail hub; workstations do not try to contact foreign machines themselves, nor do they run a *send-mail* daemon to receive incoming mail. At this time, sending mail does suffer from one of the same problems as *lpr*: mail is queued locally on the workstation if the mail hub is not responding. In such a case, there is no guarantee that it will ever be delivered; the next user may willfully or mistakenly delete any mail in the spool directory.

Notification. On a timesharing machine, it is easy to notify a user asynchronously of some event: one need simply find the entry in */etc/utmp*, if one exists, and send a message to the appropriate terminal. Where, however, does one find a user somewhere in the forest of 1000 workstations? And where does one deliver a message in a thicket of windows? To solve this problem, Athena has developed a notification system known as *Zephyr*,⁹ which can be used both to locate users and to send messages to them. One simple example is the command *zwrite treese*, which will deliver a message to user *treese* on his workstation if he is logged in somewhere on the Athena network (within constraints of permission and privacy, of course).

Remote Access. An Athena workstation is designed to be a single-user machine, if for no other reason than that it is easy to gain root access on one. By default, Athena workstations do not permit remote logins, remote shells, or remote file access, since that may harm the current user of the workstation (i.e., the one logged in on the console). A user can defeat this protection and allow access to a workstation during a session if desired. The major disadvantage of this restriction is that operations personnel cannot remotely login to repair problems; experience has shown that this is an acceptable

limitation.

On-Line Consulting. On a UNIX timesharing system, one can usually find the system manager or other knowledgeable user when one runs into problems. Finding assistance is much more difficult when there are 5000 active users of the system, with the Athena staff system wizards somewhere on the other side of the campus. To solve this problem, Athena implemented an "On-Line Consulting System" (OLC) that can be used to ask questions of consultants and other knowledgeable users logged in elsewhere on the network. Questions are saved until a consultant becomes available, and answers are often returned by electronic mail if the question remains unresolved when the user logs out.

Network Etiquette. Athena workstations must be good neighbors on the network. Services that require the use of Ethernet broadcast packets are almost always unacceptable by that measure. They raise two problems: first, the broadcasts are limited to the local net (as a matter of policy) and can therefore reach only a small fraction of the workstations. Second, they tend to generate a great deal of network traffic when many machines are involved, consuming valuable network bandwidth and local processing cycles. One example is the *rwwho/ruptime* software from 4.3BSD; handling the packets from nearly a hundred workstations on the same local net seriously affected workstation performance in the early days of workstation use at Athena. Athena machines no longer run that software.

Configuration Changes

In satisfying these constraints, Athena has made several changes to the standard UNIX configuration for use on workstations. These fall into four categories: changes to the directory hierarchy, introduction of "activated" and "deactivated" states for a workstation, modification of the login procedure, and changes to configuration files. Strictly speaking, many of these changes are not necessary for the public workstations most common at Athena, but are included for use on non-public machines, such as those in the offices of M.I.T. faculty members.

Directory structure. Most of the changes in the directory structure are in */usr*. Because some standard subdirectories are used for executable programs and some for spooling and administration, */usr* on an Athena workstation actually contains a set of symbolic links. Directories such as *adm*, *spool*, and *crash* are stored on the local disk on */site*; other directories are actually subdirectories of */urvd*, a read-only RVD system library mounted at activation.

Over time, the root filesystem has also outgrown its original size. To avoid reconfiguring all machines with a larger filesystem, many programs in */etc* or */bin* are actually symbolic links to a second RVD system library, mounted on */srvd*. These programs are those which are not essential for a workstation during

its boot procedure or for repairing a workstation in single-user mode. For example, the C compiler and the assembler fall into this category. The */srvd* system library also has the latest versions of programs that should reside on the root; it is used as a reference both for installing and for updating workstations.

Activating a Workstation. As with all systems, it is occasionally necessary to update the software available on a workstation. One limitation of read-only file service such as the RVD system is that this cannot be done while a workstation has attached the system libraries. Of course, it is possible to make the new libraries available and wait until all of the workstations have booted again to start using them. This solution takes too long to work probabilistically and so implies a visit to each workstation to boot it. This, too, is unacceptable.

Therefore, when not in use, an Athena workstation is in a "deactivated" state. No system libraries are attached and the window system is not running. In place of a *getty* on the console, a program known as *toehold* waits for a keypress from a user who wishes to login. *Toehold* then executes a shell script that attaches the system libraries. If this succeeds, *toehold* starts the X Window System, and a login window appears for the user.

After a user logs out, *toehold* "deactivates" the workstation. This includes detaching any attached filesystems, including the system libraries and filesystems that the user may have attached during his session, ensuring that remote access is impossible, cleaning the temporary storage areas (e.g., */tmp*), and killing the window system.

Toehold also has an alternate entry: typing control-P (^P) on the console allows one to login directly on the console without activating the workstation. This is particularly useful for repairing a workstation that is not working properly.

Logging In. The login process is considerably more complicated now. */bin/login* now includes the following functions:

1. It authenticates the user with *Kerberos*. To limit access on certain workstations, if the file */etc/nocreate* exists, an account is not automatically created; the user must already be listed in */etc/passwd*.
2. It adds entries for the user to */etc/passwd* and */etc/group*, using information from *Hesiod*. This makes the information available to programs that require it for the duration of the session.
3. It attaches the user's home directory on */mit/<username>*. If the directory is unavailable for some reason (e.g., the appropriate NFS server is down), a temporary home directory is created for the user in */tmp*. In this case, the user is notified of the situation and may choose to abort the login. If the file */etc/noattach* exists, the home directory is not automatically attached, and a local home directory is assumed to exist.

/bin/login then continues with the normal execution of the shell. Note that it does no longer simply *exec()* the shell; it forks before executing the shell so it can perform some cleanup operations when the user logs out. The cleanup includes deletion of *Kerberos* information and detaching the user's home directory.

As part of the login process, a *Zephyr* windowgram client is started and the *Zephyr* server informed of the login.

Configuration Files. Many UNIX configuration files have been heavily modified for workstation use; some of the most important are described here.

Boot-Time Configuration. As usual, the script */etc/rc* is executed when a workstation boots. The *rc* script on Athena workstations, however, has been extensively reworked to provide a great deal of flexibility. There are actually four files involved: *rc*, *rc.conf*, *rc.net*, and *rc.local*.

/etc/rc does most of the work, and it calls each of the other three as required. It first performs disk checks and resets the password file appropriately. It then calls *rc.conf* to obtain configuration information for the workstation and *rc.net* to initialize the network subsystem. Next, it spawns various daemons and further cleans up from the last session, including flushing all connections to file servers (both RVD and NFS). Finally, it calls *rc.local* for any workstation-specific tasks.

rc.conf sets a number of configuration variables for the workstation, including its hostname and network address. These are used to determine which daemons should be run, what configuration should be done, etc. This defines the supported set of differences between workstations and servers, and confines their specification to a single file.

rc.net performs network initialization, including configuring network interfaces, setting default routes, and starting the local nameserver. These functions are isolated in a single file to simplify starting the network in single-user mode; trying to repair a workstation often requires use of some service on the network. A program named *machtype* is used to determine the type of the machine so a single version of the file can be used on all Athena machines.

rc.local is reserved for local configuration on non-public machines; standard server configurations are handled by */etc/rc* itself.

Nameserver Configuration. Because of the *Hesiod* extensions to */etc/named*, an additional standard configuration file has been added: */etc/named.hes*. It contains the names and addresses of the *Hesiod* servers. (Note: the addresses of *Hesiod* servers are included because some software attempts to resolve names with a class ANY query. If the addresses are not present, this will result in a response that does not include the address, and the desired host cannot be contacted.) For local priming of the nameserver cache, the file */etc/named.local* is available. On public workstations it is empty.

Sendmail Configuration. The files *aliases*, *aliases.dir*, *aliases.pag*, *sendmail.cf*, and *sendmail.fc* in */usr/lib* are actually symbolic links into */site/usr/lib* to allow local configuration changes. On public workstations, the aliases files are empty, and the *sendmail.cf* file is a standard Athena version. *Sendmail.cf* is configured only to send mail, not to receive it. In addition, it rewrites "from" addresses to be from *user@ATHENA.MIT.EDU* and rewrites unqualified usernames to be to *user@ATHENA.MIT.EDU*. *Sendmail* does not run as a daemon on Athena workstations; it is started on demand to send mail and periodically by *cron* to attempt to send any queued mail.

/etc/passwd and */etc/group*. These files are updated at login time from information supplied by *Hesiod*. When a workstation deactivates, */etc/passwd.local* and */etc/group.local* are copied over these files, undoing any changes. Public workstations also initialize these files from */srvd* at boot time to prohibit other changes.

Future Plans

Implementation of the workstation environment at Project Athena is not yet finished. Some known areas of work include the following topics.

Network Error Logging. As the number of workstations and servers grows, it becomes more and more difficult to monitor what is happening. By logging error messages and status information across the network to a central machine, it may be possible to detect abnormal situations before major failures occur. The sheer scale of the system also tends to generate several occurrences of low-probability errors; monitoring them from a central location can help in understanding and solving the problem. Automatic filtering tools will be needed to cope with the volume of messages.

Automatic Software Update and Integrity Check. As noted above, updating workstation software is a major undertaking, since it currently requires a visit to each workstation. It is necessary to develop a reliable means to automatically update software on a workstation. Reliability is the key issue; a software update should not leave a trail of broken workstations. Part of this effort is a software integrity check system that verifies that the configuration and software are correct.

Shift to Vendor Software Base. As DEC and IBM produce new hardware, it will become more and more necessary for Athena to work with the vendors' software as well. This is driven partly by device support, and partly by the fact that their diverging systems make it harder to build software from nearly identical source code across different platforms. New applications also require some of the new functionality of the vendor versions. To this end, the Athena developments (e.g., *Kerberos*) must either be absorbed by the vendors or engineered as a layer above the

vendor systems. As a side effect, this engineering will enable non-Athena sites to import Athena software easily.

Dynamic Configuration. One last consequence of scale is that the configuration of a workstation should be as dynamic as possible. The names and addresses of nameservers, for example, should not be wired into a configuration file on the workstation; they should be available from a server on the local network when a machine boots. This is one exception to the use of broadcast packets above; a workstation may be permitted to broadcast an initial request for information.

In a similar vein, the Internet address of a workstation could be assigned dynamically. Already one of the most common and most annoying problems that Athena faces is the misconfigured network address. Experiments with assigning the address at boot time are already underway. One immediate application of this is for student-owned workstations: if a student moves a workstation to a different dormitory with a different subnetwork, the workstation requires a different address. The average student, however, cannot be expected to understand how to change the address or how to get a new address assigned by M.I.T. Telecommunications. Maintaining correct workstation name-to-address mappings in a dynamic environment would become much more difficult; the management issues need to be resolved as well.

Scale to 10,000 workstations/1000 servers. Athena's work at M.I.T. has generated a demand for the workstation environment to be used elsewhere on campus. This, coupled with one of Athena's original goals to eventually provide a workstation system for students to own, means that the next few years will see a dramatic increase in the number of workstations in use (and the number of servers required to support them). Scaling up another order of magnitude will require further work as we reach the limits of the work done so far. As one example, a *Hesiod* nameserver currently has an in-core image of over ten megabytes, stretching the limits of our current system configuration. This problem, of course, has a straightforward solution, but it indicates that solutions suitable for the current scale may be insufficient for the next stage.

Conclusion

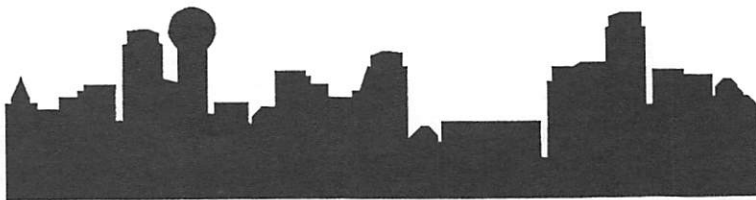
The Athena environment represents a first step in re-engineering UNIX for a distributed workstation system. As large networks become more common, the issues of scale and management will become more and more important. One wizard must suffice for perhaps a thousand workstations.

Acknowledgements

The author would like to thank Dan Geer, Kathy Lieben, Jerry Saltzer, Mark Levine, and Jennifer Steiner for valuable comments on earlier drafts of this paper. Thanks also to the staff and users of Project Athena, who have supported and endured the construction of this system.

References

1. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," in *Usenix Conference Proceedings*, (Summer, 1985).
2. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions On Graphics* 5(2) pp. 79-109 (April, 1987).
3. M. A. Rosenstein, D. E. Geer, and P. J. Levine, "Take Me To Your Leader: Service Management in a Complicated Environment," in *Usenix Conference Proceedings*, (Winter, 1988).
4. J. M. Bloom and K. J. Dunlap, "A Distributed Name Server for the DARPA Internet," pp. 172-181 in *Usenix Conference Proceedings*, (Summer, 1986).
5. S. P. Dyer, "The Hesiod Name Server," in *Usenix Conference Proceedings*, (Winter, 1988).
6. J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Usenix Conference Proceedings*, (Winter, 1988).
7. M. T. Rose, *Post Office Protocol (revised)*, University of Delaware (1985). (MH internal)
8. Rand Corp., *The Rand Message Handling System: User's Manual*, U.C.I. Dept. of Information & Computer Science, Irvine, California (November, 1985).
9. C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, "The Zephyr Notification System," in *Usenix Conference Proceedings*, (Winter, 1988).



The *Hesiod* Name Server

Stephen P. Dyer
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
dyer@ATHENA.MIT.EDU

ABSTRACT

Hesiod[†], the Athena name server, provides naming for services and data objects in a distributed network environment. More specifically, it replaces databases that heretofore have had to be duplicated on each workstation and timesharing machine (e.g., remote file system information, */etc/printcap*, */etc/services*, */etc/passwd*, */etc/group*) and provides a flexible mechanism to supply new information as the need arises.

Introduction and Purpose

The computing environment at Project Athena has recently changed from a group of timesharing machines to a collection of file servers and many hundreds (and potentially many thousands) of publically-accessible workstations. The origins of UNIX as a time-sharing system become acutely obvious when confronted with the need to manage information for hundreds of machines that may be used by many different individuals. The method used by UNIX to maintain information for its users and programs has been ASCII database files stored on each machine which are authoritative for all users of that machine. However, this breaks down when the number of machines and potential users are multiplied by two or three orders of magnitude. The system management effort to keep each machine's information current grows directly as the number of machines; this quickly becomes unworkable with more than a few dozen machines. We wanted a solution that could easily accommodate Athena's expected growth for the next 5 years.

Rather than having information duplicated on each machine, the concept of retrieving information via a network service, a *name server*, has proved workable and reliable. Xerox's *Clearinghouse*,¹ Sun's *Yellow Pages*,² and the Internet Domain Name Server^{3,4} are examples of name services in current use. We chose to base our name service, *Hesiod*, on the Berkeley Internet Domain Name Server, BIND,⁵ for several reasons. First, the design had proved itself through its use in the Internet over the past several years, and it had a number of features that made it an attractive base for *Hesiod*: its hierarchical name space, the ability to delegate authority to subsidiary name servers, and the ability to take advantage of local caching of data to improve performance.

Second, the BIND source code was readily available and provided a firm foundation for a more general name service; we did not have to spend time building low-level support facilities which it already provided. Finally, BIND source code is non-proprietary, which would facilitate our distribution of *Hesiod* to other interested sites.

Hesiod provides a name service for use by workstations and timesharing systems. It does not address the problems of centralized management and distribution of such information, which is provided by another service, the Athena Service Management System, or SMS.⁶ SMS maintains and distributes information managed by Athena Operations to each of the Athena *Hesiod* name servers. *Hesiod* may be used without SMS; neither is dependent on the other. However, without an information management system front end, the *Hesiod* databases are simply ASCII files in BIND-compatible resource records format that must then be managed with a text editor. Large sites may appreciate the convenience SMS provides, while smaller sites may opt for the simplicity of using *Hesiod* without SMS.

Hesiod provides a content-addressable memory where certain strings can be mapped to others depending on the query. *Hesiod* has no knowledge about the data it stores; queries and responses are simple key/content interactions. It is designed to be used in situations where a small amount of data that changes infrequently needs to be retrieved quickly, with little overhead. It is not intended to serve as a general-purpose database system supporting arbitrary queries, or as a repository for information that changes frequently. The current implementation provides no facility for an arbitrary application to update the *Hesiod* database, which is refreshed several times a day by the Athena SMS. Because of the limitation imposed by the underlying implementation of *Hesiod*, based as it is on the Internet domain naming scheme, there is a maximum length of 512 bytes of data that

[†]n. 8th century B.C. Greek poet. The names of the Gods and the myths surrounding them are recorded in his poetry.

can be exchanged between the client and the name servers using UDP datagrams. This imposes limits on both the maximum size of an individual data record, as well as the number of records that can be returned in a single packet in the case of multiple matches. *Hesiod* was designed to provide applications with a rapid, low-overhead naming service in which a query would return no more than a few matches of limited size. Applications that require more complicated queries or ones that return voluminous data should consider interfacing to SMS.

Hesiod Queries

A *Hesiod* query consists of two parts, a *HesiodName*, which is the name of an object in the network, and a *HesiodNameType*, an application-specific qualifier that identifies the application space in which that object is named.

We do not use standard Internet Domain Name notation to refer to *HesiodNames* for several reasons: First, we wish to have objects with name containing the '.' character.¹ In Internet domain notation, a name that contains a '.' is considered fully resolved. Second, early BIND implementations had no provision for deciding the proper domain suffix to use when resolving a relative name.

A name given to the *Hesiod* name server for resolution looks like:

```
HESIODNAME => LHS
HESIODNAME => LHS@RHS
LHS => [Any ASCII character, except NUL and '@']*
      { 0 or more characters from this alphabet }
RHS => [Any ASCII character, except NUL and '@']+
      { 1 or more characters from this alphabet }
```

In other words, a *HesiodName* consists of [LHS][@RHS] where either [LHS] or [@RHS] need not be present.

The LHS of a *HesiodName* is *uninterpreted*; although it may be modified according to the rules described by the information in */etc/hesiod.conf* (see below), it is not itself a domain name.

We define a set of routines known as the *Hesiod* library that take two strings, a *HesiodName* and a user-supplied key, a *HesiodNameType*, convert it to a fully-qualified domain name, call the BIND library, and return the results to the original caller. The *HesiodNameType* is a well-known string that is provided by an application that uses the *Hesiod* library. It is used directly in the expansion of a *Hesiod* name to a BIND name (see below) without further indirection or translation. A new *HesiodNameType* comes into existence simply by being used by an application; no libraries or configuration files need to be modified. Naturally, there has to be appropriate data stored by the name server which is associated with that *HesiodNameType*.

¹As just one example, MIT course names, such as 6.001, contain periods.

To provide an example, one of the routines in the *Hesiod* library takes a *HesiodName* and returns a fully-qualified name to be handed to BIND:

```
char *
hes_to_bind(HesiodName, HesiodNameType)
char *HesiodName, *HesiodNameType;
```

The *HesiodNameType* identifies the query to make to BIND and the proper expansion rules to use with the LHS and RHS of the name. This would be chosen by the application, and could be application-specific.

Thus, the following are valid *HesiodNames*:

```
14.21
default-printer
default-printer@SIPB
@heracles
@heracles.MIT.EDU
kerberos@Berkeley.MIT.EDU
```

The configuration file */etc/hesiod.conf* contains two tables specifying the treatment of LHS and RHS components of a *HesiodName*. In the translation of a *HesiodName* to a valid BIND name, the LHS is expanded by concatenating together the *HesiodName*, the separator '.', the *HesiodNameType*, and the LHS entry found in the configuration files. If the RHS is null, the RHS entry in the configuration file is used. If the RHS is a fully qualified domain name already, it is used directly. Otherwise, if a RHS is present, it is used as a *HesiodName* for further resolution against the *HesiodNameType*, "rhs-extension". If this query succeeds, the first reply is used as the RHS, otherwise an error is returned. The fully-expanded LHS and RHS are then concatenated together, separated by a '.', and this value is passed to BIND for resolution.

The following is a typical copy of */etc/hesiod.conf*:

```
#file /etc/hesiod.conf
#comment lines begin with a '#' in column 1
#LHS table
lhs = .ns
#RHS table
rhs = .Athena.MIT.EDU
```

With this definition, a call to *hes_to_bind* ("e40", "printer") would produce a LHS of "e40.printer.ns" and a RHS of ".athena.MIT.EDU", and the resulting BIND name, "e40.printer.ns.Athena.MIT.EDU".

In C pseudo-code, we would have the following productions:

```
hes_to_bind("14.21, "filesystem") =>
    "14.21.filesys.ns.Athena.MIT.EDU"
hes_to_bind("e40, "printer") =>
    "e40.printer.ns.Athena.MIT.EDU"
hes_to_bind("SIPB, "rhs-extension") =>
    "SIPB.rhs-extension.ns.Athena.MIT.EDU"
hes_to_bind("default@SIPB, "printer") =>
    "default.printer.ns.SIPB.MIT.EDU"
```

```
(this assumes that the previous
production resolved to
"SIPB.MIT.EDU")
hes_to_bind("kerberos@Berkeley.EDU", "sloc")
=> "kerberos.sloc.ns.Berkeley.EDU"
```

These productions are then passed to the BIND name server for resolution.

Data Types

Hesiod data are stored as Internet domain resource records. A new class, HS, signifying a *Hesiod* query or datum has been reserved, and a new query type, TXT, that allows the storage of arbitrary ASCII strings. Paul Mockapetris, the Internet Domain System designer, has recently specified the HS class and the TXT type in RFCs 1034 and 1035.

BIND Requirements

A version of BIND that supports the HS query class and TXT query type is required to support the *Hesiod* name service. The latest release of BIND as of 12/31/1987, version 4.7.3, has been modified at Athena to support this, and we will be forwarding these changes to Berkeley for future releases of BIND.

Athena Client Applications of *Hesiod*

Many applications and subroutines have been modified to take advantage of the *Hesiod* service. See Appendix A for an enumeration of some of the *HesiodNameTypes* in common use within Project Athena.

The *attach* program queries *Hesiod* for the filesystem with the given name, retrieves the data, and mounts the appropriate RVD or NFS⁷ filesystems, while also authenticating the user to the file server using *Kerberos*.⁸

Login uses the user's login name as a *HesiodName* to retrieve the user's */etc/passwd* and group membership information. The actual password field is not used; rather the *Kerberos* service authenticates the user. *Login* queries *Hesiod* to determine which *Kerberos* server to invoke. By convention, the username is also the name of the user's default filesystem. The *login* program runs the *attach* program (*q.v.*) with the user's login name as an argument to mount the user's home directory.

Athena users receive their mail on POP⁹ (post-office protocol) servers. We have modified the MH programs *inc* and *msgchk* to query *Hesiod* for the location of the user's POP server.

The *lpr* program is compiled with a special version of the */etc/printcap* access library that queries *Hesiod* if the printer name cannot be found in the local */etc/printcap*.

There are optional implementations of *getpwnam()*, *getgrnam()*, and their inverse counterparts that query *Hesiod* for name-to-UID, name-

to-GID translation, and vice-versa. The same library includes an implementation of *getservent()* that queries *Hesiod* in preference to lookup in the file */etc/services*.

Hesiod Resource Records and Data Files

Appendix B lists samples of the resource records that we store on behalf of *Hesiod* client applications. The format of the ASCII strings returned by *Hesiod* is application-specific. In the case of queries that have an inverse operation, such as queries with the *HesiodNameTypes*, *passwd*, and *uid*, the *uid* resource records are *CNAMEs* for the corresponding *passwd* records.

The BIND boot file on each workstation, */etc/named.boot*, refers to an auxiliary cache file, */etc/named.hes*, that specifies the authoritative name servers for *Hesiod* queries.

Programming with the *Hesiod* Library

There are only two subroutines, *hes_resolve()* and *hes_error()*, that are usually invoked by the applications programmer when using *Hesiod*. The subroutine *hes_resolve()* is the primary interface into the *Hesiod* name server. It takes two string arguments, the name to be resolved, the *HesiodName*, and a type indicating the type of service associated with this name, the *HesiodNameType*. *hes_resolve()* returns a pointer to an array of strings, much like *argv[]*, containing all the data that matched the query, one match per array slot. The array is NULL terminated. A second call to *hes_resolve()* will overwrite any previously-returned data, so applications that require data to be maintained across multiple calls to *hes_resolve()* should copy the returned values into data areas they maintain.

Note that a call to *hes_resolve()* may return more than one match. The semantics of using or choosing between multiple matches is dependent on the particular application. In general, however, multiple matches are considered "equivalent", and any of them could be used equally well. This is exploited, for example, by the *attach* command that attaches a remote file system to the workstation. In the case of system libraries, multiple copies of which are considered equivalent, the *attach* command iterates through all matches, stopping after the first successful attach. Because *Hesiod* is based on the Internet Domain Naming scheme, no interpretation can or should be given to the order in which matches are returned.

If *hes_resolve()* returns NULL, then no data could be found, either because the name server had no matching records or an error occurred. The function *hes_error()* takes no arguments and returns a small integer indicating the type of error, if any, encountered in the last call to *hes_resolve()*.

It is important to emphasize that *Hesiod* knows nothing about the data it stores; any meaning given

to the *HesiodName*, the *HesiodNameType* and the data returned by *Hesiod* is completely imposed by the application. The format of the data stored by *Hesiod* is application-specific, and would be defined by the application programmer.

```
#include <hesiod.h>

char *HesiodName, *HesiodNameType;
char **hp;

hp = hes_resolve(HesiodName,
                 HesiodNameType);
if (hp == NULL) {
    err = hes_error();
    switch(err) {
        .
        .
        .
    }
} else {
    /* do your thing with hp */
    while(*hp != NULL) process(*hp++);
}
```

The error values returned by *hes_error()* are one of the following:

```
#define HES_ER_UNINIT      -1
#define HES_ER_OK          0
#define HES_ER_NOTFOUND    1
#define HES_ER_CONFIG      2
#define HES_ER_NET         3
```

The most common values returned by *hes_error()* are *HES_ER_OK*, meaning no error, and *HES_ER_NOTFOUND*, meaning that the desired name was not found in the *Hesiod* data base. *HES_ER_CONFIG* indicates a problem with the optional per-machine *Hesiod* configuration file, */etc/hesiod.conf*. *HES_ER_UNINIT* will never be returned by *hes_error()*, unless it is called before the first time *hes_resolve()* is called. *HES_ER_NET* indicates that the request never received a response from the *Hesiod* name server. This can be due to a variety of network problems: for example, the host making the request might be disconnected from the network, an intervening gateway might be down, or no *Hesiod* name servers responded. No further information about the state of the network is available because the domain system on which *Hesiod* is based uses datagrams with retries as the communications interface.

HES_ER_NOTFOUND is a negative acknowledgement indicating that the desired name/*HesiodNameType* pair was not found in the *Hesiod* database. An application receiving this error message can consider this an authoritative response. Of course, this may be due to an omission in the database, or simply reflect a delay between the time *Hesiod* data was asked to be placed into the database, and the actual *Hesiod* updates, which occur several times each day.

In the case of a *Hesiod* error of *HES_ER_NET*, it may be prudent for an application to assume that this situation is temporary, and that a later call to *hes_resolve()* will either return the desired data or a definitive reply of *HES_ER_NOTFOUND*. *HES_ER_CONFIG* indicates a problem with the *Hesiod* configuration file, a situation that requires intervention by a wizard and will not resolve itself spontaneously. Because no query to the *Hesiod* name server is actually made, no conclusion can be drawn about the validity of the name to be resolved. The standard Athena distribution of the *Hesiod* library does not require a configuration file; its built-in defaults suffice, so this situation should not be encountered frequently.

A general design strategy for applications using *Hesiod* is to have a contingency plan in place in case *Hesiod* does not respond, is configured incorrectly or does not know the name. This may be built-in to the application, such as new versions of *lpr* that revert to using the old printcap libraries if *Hesiod* printer information is not available. Another popular scheme, exploited by the *MH* application *inc* and the EMACS tool, *movemail*, is to allow the value of an environment variable, in this case, *MAILHOST*, to override the call to *Hesiod* to retrieve a person's mailhost, using his username as the key. Thus, a user can temporarily "hard-wire" appropriate values to allow applications to proceed. Not every application can be programmed in such a fashion, but it is prudent to try to design applications with this in mind.

Database Size and Performance

A measure of how successful *Hesiod* has been in its deployment over the past six months is how infrequently problems have appeared. For the most part, applications make *Hesiod* queries and receive answers with millisecond delays. Today, the *Hesiod* database for Project Athena contains almost three megabytes of data: roughly 9500 */etc/passwd* entries, 10000 */etc/group* entries, 6500 file system entries and 8600 post office records. There are three primary *Hesiod* nameservers distributed across the campus network.

BIND has proven itself remarkably robust in accommodating such a large, monolithic database. One problem has been noticed: the time to load the primary nameservers (which are updated from the Athena SMS every six hours) has increased markedly as the size of our data has grown. At this point, it takes approximately 20 minutes to reload a primary nameserver running on a VAX 750 and each primary nameserver's working set is approximately 10 megabytes. By staggering the times to reload each of the primary *Hesiod* servers, this has not proved to be a large operational problem. However, it does point out an area that should be examined for improved performance. Because the *HesiodNameType* component in the domain name passed to BIND identifies a potentially separate start of authority, the *Hesiod* database could be split across two or more primary

nameservers, each authoritative for a subset of the full database. This would reduce the time to load each nameserver and the size of its working set.

Acknowledgements

Jerry Saltzer, Technical Director of Project Athena, provided a great deal of assistance and guidance in the design of the *Hesiod* name server. Thanks, too, go to Dan Geer and Jeff Schiller for their assistance during the design and deployment stages. Clifford Neuman, presently at the University of Washington, and Felix Hsu of Digital Equipment Corporation participated in the early design of the system.

References

1. Xerox Corporation, *Clearinghouse Protocol*. April, 1984.
2. Sun Microsystems, *Yellow Pages Protocol Specification*. 1986.
3. P. Mockapetris, *RFC 1034 - Domain Names - Concepts and Facilities*, USC/Information Sciences Institute (November, 1987).
4. P. Mockapetris, *RFC 1035 - Domain Implementation and Specification*, USC/Information Sciences Institute (November, 1987).
5. J. M. Bloom and K. J. Dunlap, "A Distributed Name Server for the DARPA Internet," pp. 172-181 in *Usenix Conference Proceedings*, (Summer, 1986).
6. M. A. Rosenstein, D. E. Geer, and P. J. Levine, "Take Me To Your Leader: Service Management in a Complicated Environment," in *Usenix Conference Proceedings*, (Winter, 1988).
7. Sun Microsystems, *NFS Protocol Specification and Services Manual*, Revision A 1987.
8. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Section E.2.1: Kerberos Authentication and Authorization System*, M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).
9. M. T. Rose, *Post Office Protocol (revised)*, University of Delaware (1985). (MH internal)

Appendix A: *Hesiod* Name Types in Use at Athena

Table 1 shows a list of some of the presently-defined *Hesiod* Name Types, the type of information provided as a *Hesiod* Name, and the applications programs that use such queries.

<i>Hesiod</i> Name	<i>Hesiod</i> Name Type	Used By	Info Returned
workstation name	"cluster"	<i>getcluster</i>	workstation cluster information
filesystem name	"filsys"	<i>attach/detach</i>	RVD and NFS file system info
username	"pobox"	MH <i>inc/movemail</i>	location and type of mailbox
username	"passwd"	<i>tochold/login</i>	Athena-wide /etc/passwd entry
uid (ASCII)	"uid"	<i>getpwent()</i> , <i>et. al.</i>	Athena-wide UID to username mapping
group name	"group"	<i>getpwent()</i> , <i>et. al.</i>	Athena-wide /etc/group entry (no membership list)
group name	"grplist"	<i>getgrnt()</i> , <i>et. al.</i>	Athena-wide group membership mapping
gid (ASCII)	"gid"	<i>getgrnt()</i> , <i>et. al.</i>	Athena-wide GID to group name mapping
printer name	"pcap"	<i>pgetent()</i>	Athena-wide /etc/printcap entry
service name	"service"	<i>getservent()</i>	Athena-wide /etc/services entry
service name	"sloc"	<i>On-Line Consulting (OLC)</i> <i>Kerberos</i>	Host name to contact for this service (for those services that do not reside on every host)

Table 1

Appendix B – Sample Resource Records from Current *Hesiod* Database Files

```
# filsys.db
# format of data is
#      filesystem-type name-on-server server-hostname mount-mode mount-point
dyer.filsys      HS      TXT "NFS /mit/dyer eurydice w /mit/dyer"
dyfeigen.filsys HS      TXT "NFS /mit/lockers/dyfeigen zeus w /mit/dyfeigen"
dyim.filsys      HS      TXT "NFS /mit/lockers/dyim zeus w /mit/dyim"
bldg1-rtsys.filsys HS      TXT "RVD rtsys oath r /srvd"
bldg1-rtsys.filsys HS      TXT "RVD rtsys persephone r /srvd"

# gid.db
# format of data is
#      canonical name with this group id
481.gid HS      CNAME 10.01.group
483.gid HS      CNAME 10.01a.group
484.gid HS      CNAME 10.01b.group
639.gid HS      CNAME 10.01sa.group
640.gid HS      CNAME 10.01sb.group
638.gid HS      CNAME 10.01t.group

# group.db
# format of data is
#      /etc/group entry
10.01.group HS      TXT 10.01:*:481:
10.01a.group HS      TXT 10.01a:*:483:
10.01b.group HS      TXT 10.01b:*:484:
10.01sa.group HS      TXT 10.01sa:*:639:
10.01sb.group HS      TXT 10.01sb:*:640:
10.01t.group HS      TXT 10.01t:*:638:

# grplist.db
# format of data is
#      groupname1:gid1:groupname2:gid2:...
10.01.grplist HS      TXT "10.01:481:10.01t:638"
10.01ta.grplist HS      TXT "10.01t:638"

# passwd.db
# format of data is
#      /etc/passwd entry
dyer.passwd HS      TXT "dyer:*:17287:101:Steve Dyer,,,:/mit/dyer:/bin/csh"

# pobox.db
# format of data is
#      post-office-type server-host-name mailbox-name
dyer.pobox HS      TXT "POP E40-PO.MIT.EDU dyer"

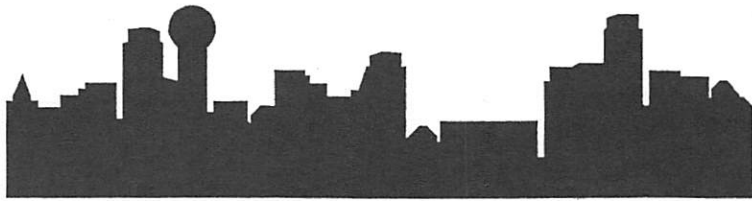
# printcap.db
# format of data is
#      /etc/printcap entry
nil.pcap HS      TXT "nil|LPS-40:rp=nil:rm=castor.mit.edu:sd=/usr/spool/printer/nil:"

# service.db
# format of data is
#      service-name protocol port-number
discard.service HS      TXT "discard tcp 9"
discard.service HS      TXT "discard udp 9"
nntp.service HS      TXT "nntp tcp 119"
```

```
# sloc.db
# format of data is
#      host name where this service is offered
zephyr.sloc      HS      TXT ARILINN.MIT.EDU
zephyr.sloc      HS      TXT NESKAYA.MIT.EDU
zephyr.sloc      HS      TXT ORPHEUS.MIT.EDU
zephyr.sloc      HS      TXT PRIAM.MIT.EDU
```

```
# uid.db
# format of data is
#      canonical name with this user id
17287.uid        HS      CNAME  dyer.passwd
```





USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

Kerberos: An Authentication Service for Open Network Systems

Jennifer G. Steiner
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
steiner@ATHENA.MIT.EDU

Clifford Neuman¹
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195
bcn@CS.WASHINGTON.EDU

Jeffrey I. Schiller
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
jis@ATHENA.MIT.EDU



ABSTRACT

In an open network computing environment, a workstation cannot be trusted to identify its users correctly to network services. *Kerberos* provides an alternative approach whereby a trusted third-party authentication service is used to verify users' identities. This paper gives an overview of the *Kerberos* authentication model as implemented for MIT's Project Athena. It describes the protocols used by clients, servers, and *Kerberos* to achieve authentication. It also describes the management and replication of the database required. The views of *Kerberos* as seen by the user, programmer, and administrator are described. Finally, the role of *Kerberos* in the larger Athena picture is given, along with a list of applications that presently use *Kerberos* for user authentication. We describe the addition of *Kerberos* authentication to the Sun Network File System as a case study for integrating *Kerberos* with an existing application.

Introduction

This paper gives an overview of *Kerberos*, an authentication system designed by Miller and Neuman¹ for open network computing environments, and describes our experience using it at MIT's Project Athena.² In the first section of the paper, we explain why a new authentication model is needed for open networks, and what its requirements are. The second section lists the components of the *Kerberos* software and describes how they interact in providing the authentication service. In Section 3, we describe the *Kerberos* naming scheme.

Section 4 presents the building blocks of *Kerberos* authentication — the *ticket* and the

authenticator. This leads to a discussion of the two authentication protocols: the initial authentication of a user to *Kerberos* (analogous to logging in), and the protocol for mutual authentication of a potential consumer and a potential producer of a network service.

Kerberos requires a database of information about its clients; Section 5 describes the database, its management, and the protocol for its modification. Section 6 describes the *Kerberos* interface to its users, applications programmers, and administrators. In Section 7, we describe how the Project Athena *Kerberos* fits into the rest of the Athena environment. We also describe the interaction of different *Kerberos* authentication domains, or *realms*; in our case, the relation between the Project Athena *Kerberos* and the *Kerberos* running at MIT's Laboratory for Computer Science.

¹Clifford Neuman was a member of the Project Athena staff during the design and initial implementation phase of *Kerberos*.

In Section 8, we mention open issues and problems as yet unsolved. The last section gives the current status of *Kerberos* at Project Athena. In the appendix, we describe in detail how *Kerberos* is applied to a network file service to authenticate users who wish to gain access to remote file systems.

Conventions

Throughout this paper we use terms that may be ambiguous, new to the reader, or used differently elsewhere. Below we state our use of those terms.

User, Client, Server. By *user*, we mean a human being who uses a program or service. A *client* also uses something, but is not necessarily a person; it can be a program. Often network applications consist of two parts; one program which runs on one machine and requests a remote service, and another program which runs on the remote machine and performs that service. We call those the *client* side and *server* side of the application, respectively. Often, a *client* will contact a *server* on behalf of a *user*.

Each entity that uses the *Kerberos* system, be it a user or a network server, is in one sense a client, since it uses the *Kerberos* service. So to distinguish *Kerberos* clients from clients of other services, we use the term *principal* to indicate such an entity. Note that a *Kerberos* principal can be either a user or a server. (We describe the naming of *Kerberos* principals in a later section.)

Service vs. Server. We use *service* as an abstract specification of some actions to be performed. A process which performs those actions is called a *server*. At a given time, there may be several *servers* (usually running on different machines) performing a given *service*. For example, at Athena there is one BSD UNIX *rlogin* server running on each of our timesharing machines.

Key, Private Key, Password. *Kerberos* uses private key encryption. Each *Kerberos* principal is assigned a large number, its private key, known only to that principal and *Kerberos*. In the case of a user, the private key is the result of a one-way function applied to the user's *password*. We use *key* as shorthand for *private key*.

Credentials. Unfortunately, this word has a special meaning for both the Sun Network File System and the *Kerberos* system. We explicitly state whether we mean NFS credentials or *Kerberos* credentials, otherwise the term is used in the normal English language sense.

Master and Slave. It is possible to run *Kerberos* authentication software on more than one machine. However, there is always only one definitive copy of the *Kerberos* database. The machine which houses this database is called the *master* machine, or just the *master*. Other machines may possess read-only copies of the *Kerberos* database, and these are called *slaves*.

Motivation

In a non-networked personal computing environment, resources and information can be protected by physically securing the personal computer. In a timesharing computing environment, the operating system protects users from one another and controls resources. In order to determine what each user is able to read or modify, it is necessary for the timesharing system to identify each user. This is accomplished when the user logs in.

In a network of users requiring services from many separate computers, there are three approaches one can take to access control: One can do nothing, relying on the machine to which the user is logged in to prevent unauthorized access; one can require the host to prove its identity, but trust the host's word as to who the user is; or one can require the user to prove her/his identity for each required service.

In a closed environment where all the machines are under strict control, one can use the first approach. When the organization controls all the hosts communicating over the network, this is a reasonable approach.

In a more open environment, one might selectively trust only those hosts under organizational control. In this case, each host must be required to prove its identity. The *rlogin* and *rsh* programs use this approach. In those protocols, authentication is done by checking the Internet address from which a connection has been established.

In the Athena environment, we must be able to honor requests from hosts that are not under organizational control. Users have complete control of their workstations: they can reboot them, bring them up standalone, or even boot off their own tapes. As such, the third approach must be taken; the user must prove her/his identity for each desired service. The server must also prove its identity. It is not sufficient to physically secure the host running a network server; someone elsewhere on the network may be masquerading as the given server.

Our environment places several requirements on an identification mechanism. First, it must be secure. Circumventing it must be difficult enough that a potential attacker does not find the authentication mechanism to be the weak link. Someone watching the network should not be able to obtain the information necessary to impersonate another user. Second, it must be reliable. Access to many services will depend on the authentication service. If it is not reliable, the system of services as a whole will not be. Third, it should be transparent. Ideally, the user should not be aware of authentication taking place. Finally, it should be scalable. Many systems can communicate with Athena hosts. Not all of these will support our mechanism, but software should not break if they did.

Kerberos is the result of our work to satisfy the above requirements. When a user walks up to a

workstation s/he "logs in". As far as the user can tell, this initial identification is sufficient to prove her/his identity to all the required network servers for the duration of the login session. The security of *Kerberos* relies on the security of several authentication servers, but not on the system from which users log in, nor on the security of the end servers that will be used. The authentication server provides a properly authenticated user with a way to prove her/his identity to servers scattered across the network.

Authentication is a fundamental building block for a secure networked environment. If, for example, a server knows for certain the identity of a client, it can decide whether to provide the service, whether the user should be given special privileges, who should receive the bill for the service, and so forth. In other words, authorization and accounting schemes can be built on top of the authentication that *Kerberos* provides, resulting in equivalent security to the lone personal computer or the timesharing system.

What is Kerberos?

Kerberos is a trusted third-party authentication service based on the model presented by Needham and Schroeder.³ It is trusted in the sense that each of its clients believes *Kerberos*' judgement as to the identity of each of its other clients to be accurate. Time-stamps (large numbers representing the current date and time) have been added to the original model to aid in the detection of *replay*. Replay occurs when a message is stolen off the network and resent later. For a more complete description of replay, and other issues of authentication, see Voydock and Kent.⁴

What Does It Do?

Kerberos keeps a database of its clients and their *private keys*. The private key is a large number known only to *Kerberos* and the client it belongs to. In the case that the client is a user, it is an encrypted password. Network services requiring authentication register with *Kerberos*, as do clients wishing to use those services. The private keys are negotiated at registration.

Because *Kerberos* knows these private keys, it can create messages which convince one client that another is really who it claims to be. *Kerberos* also generates temporary private keys, called *session keys*, which are given to two clients and no one else. A session key can be used to encrypt messages between two parties.

Kerberos provides three distinct levels of protection. The application programmer determines which is appropriate, according to the requirements of the application. For example, some applications require only that authenticity be established at the initiation of a network connection, and can assume that further messages from a given network address originate from the authenticated party. Our authenticated network file system uses this level of security.

Other applications require authentication of each

message, but do not care whether the content of the message is disclosed or not. For these, *Kerberos* provides *safe messages*. Yet a higher level of security is provided by *private messages*, where each message is not only authenticated, but also encrypted. Private messages are used, for example, by the *Kerberos* server itself for sending passwords over the network.

Software Components

The Athena implementation comprises several modules (see Figure 1). The *Kerberos* applications library provides an interface for application clients and application servers. It contains, among others, routines for creating or reading authentication requests, and the routines for creating safe or private messages.

- *Kerberos* applications library
- encryption library
- database library
- database administration programs
- administration server
- authentication server
- db propagation software
- user programs
- applications

Figure 1. *Kerberos* Software Components.

Encryption in *Kerberos* is based on DES, the Data Encryption Standard.⁵ The encryption library implements those routines. Several methods of encryption are provided, with tradeoffs between speed and security. An extension to the DES Cypher Block Chaining (CBC) mode, called the Propagating CBC mode, is also provided. In CBC, an error is propagated only through the current block of the cipher, whereas in PCBC, the error is propagated throughout the message. This renders the entire message useless if an error occurs, rather than just a portion of it. The encryption library is an independent module, and may be replaced with other DES implementations or a different encryption library.

Another replaceable module is the database management system. The current Athena implementation of the database library uses *ndbm*, although Ingres was originally used. Other database management libraries could be used as well.

The *Kerberos* database needs are straightforward; a record is held for each principal, containing the name, private key, and expiration date of the principal, along with some administrative information. (The expiration date is the date after which an entry is no longer valid. It is usually set to a few years into the future at registration.)

Other user information, such as real name, phone number, and so forth, is kept by another server, the *Hesiod* nameserver.⁶ This way, sensitive information, namely passwords, can be handled by *Kerberos*, using fairly high security measures; while

the non-sensitive information kept by *Hesiod* is dealt with differently; it can, for example, be sent unencrypted over the network.

The *Kerberos* servers use the database library, as do the tools for administering the database.

The *administration server* (or *KDBM server*) provides a read-write network interface to the database. The client side of the program may be run on any machine on the network. The server side, however, must run on the machine housing the *Kerberos* database in order to make changes to the database.

The *authentication server* (or *Kerberos server*), on the other hand, performs read-only operations on the *Kerberos* database, namely, the authentication of principals, and generation of session keys. Since this server does not modify the *Kerberos* database, it may run on a machine housing a read-only copy of the master *Kerberos* database.

Database propagation software manages replication of the *Kerberos* database. It is possible to have copies of the database on several different machines, with a copy of the authentication server running on each machine. Each of these *slave* machines receives an update of the *Kerberos* database from the *master* machine at given intervals.

Finally, there are end-user programs for logging in to *Kerberos*, changing a *Kerberos* password, and displaying or destroying *Kerberos tickets* (tickets are explained later on).

Kerberos Names

Part of authenticating an entity is naming it. The process of authentication is the verification that the client is the one named in a request. What does a name consist of? In *Kerberos*, both users and servers are named. As far as the authentication server is concerned, they are equivalent. A name consists of a primary name, an instance, and a realm, expressed as *name.instance@realm* (see Figure 2).

```
bcn
treese.root
jis@LCS.MIT.EDU
rlogin.priam@ATHENA.MIT.EDU
```

Figure 2. *Kerberos* Names.

The *primary name* is the name of the user or the service. The *instance* is used to distinguish among variations on the primary name. For users, an instance may entail special privileges, such as the "root" or "admin" instances. For services in the Athena environment, the instance is usually the name of the machine on which the server runs. For example, the *rlogin* service has different instances on different hosts: *rlogin.priam* is the *rlogin* server on the host named *priam*. A *Kerberos* ticket is only good for a single named server. As such, a separate ticket is required to gain access to different instances of the

same service. The *realm* is the name of an administrative entity that maintains authentication data. For example, different institutions may each have their own *Kerberos* machine, housing a different database. They have different *Kerberos* realms. (Realms are discussed further in section 8.2.)

How It Works

This section describes the *Kerberos* authentication protocols. The following abbreviations are used

c	->	client
s	->	server
addr	->	client's network address
life	->	lifetime of ticket
tgs, TGS	->	ticket-granting server
Kerberos	->	authentication server
KDBM	->	administration server
K_x	->	x's private key
$K_{x,y}$	->	session key for x and y
$\{abc\}K_x$	->	abc encrypted in x's key
$T_{x,y}$	->	x's ticket to use y
A_x	->	authenticator for x
WS	->	workstation

in the figures.

As mentioned above, the *Kerberos* authentication model is based on the Needham and Schroeder key distribution protocol. When a user requests a service, her/his identity must be established. To do this, a ticket is presented to the server, along with proof that the ticket was originally issued to the user, not stolen. There are three phases to authentication through *Kerberos*. In the first phase, the user obtains credentials to be used to request access to other services. In the second phase, the user requests authentication for a specific service. In the final phase, the user presents those credentials to the end server.

Credentials

There are two types of credentials used in the *Kerberos* authentication model: *tickets* and *authenticators*. Both are based on private key encryption, but they are encrypted using different keys. A ticket is used to securely pass the identity of the person to whom the ticket was issued between the authentication server and the end server. A ticket also passes information that can be used to make sure that the person using the ticket is the same person to which it was issued. The authenticator contains the additional information which, when compared against that in the ticket proves that the client presenting the ticket is the same one to which the ticket was issued.

A ticket is good for a single server and a single client. It contains the name of the server, the name of the client, the Internet address of the client, a timestamp, a lifetime, and a random session key. This information is encrypted using the key of the server

for which the ticket will be used. Once the ticket has been issued, it may be used multiple times by the named client to gain access to the named server, until the ticket expires. Note that because the ticket is encrypted in the key of the server, it is safe to allow the user to pass the ticket on to the server without having to worry about the user modifying the ticket (see Figure 3).

$\{s, c, \text{addr}, \text{timestamp}, \text{life}, K_{s,c}\}K_s$

Figure 3. A Kerberos Ticket.

Unlike the ticket, the authenticator can only be used once. A new one must be generated each time a client wants to use a service. This does not present a problem because the client is able to build the authenticator itself. An authenticator contains the name of the client, the workstation's IP address, and the current workstation time. The authenticator is encrypted in the session key that is part of the ticket (see Figure 4).

$\{c, \text{addr}, \text{timestamp}\}K_{s,c}$

Figure 4. A Kerberos Authenticator.

Getting the Initial Ticket

When the user walks up to a workstation, only one piece of information can prove her/his identity: the user's password. The initial exchange with the authentication server is designed to minimize the chance that the password will be compromised, while at the same time not allowing a user to properly authenticate her/himself without knowledge of that password. The process of logging in appears to the user to be the same as logging in to a timesharing system. Behind the scenes, though, it is quite different (see Figure 5).

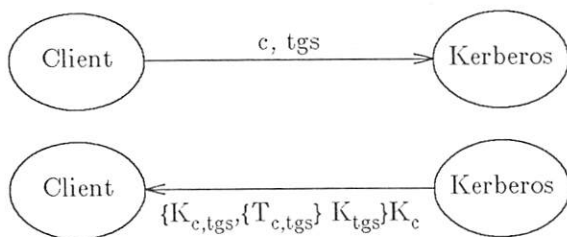


Figure 5. Getting the Initial Ticket.

The user is prompted for her/his username. Once it has been entered, a request is sent to the authentication server containing the user's name and the name of a special service known as the *ticket-granting service*.

The authentication server checks that it knows about the client. If so, it generates a random session key which will later be used between the client and the ticket-granting server. It then creates a ticket for the ticket-granting server which contains the client's name, the name of the ticket-granting server, the current time, a lifetime for the ticket, the client's IP

address, and the random session key just created. This is all encrypted in a key known only to the ticket-granting server and the authentication server.

The authentication server then sends the ticket, along with a copy of the random session key and some additional information, back to the client. This response is encrypted in the client's private key, known only to *Kerberos* and the client, which is derived from the user's password.

Once the response has been received by the client, the user is asked for her/his password. The password is converted to a DES key and used to decrypt the response from the authentication server. The ticket and the session key, along with some of the other information, are stored for future use, and the user's password and DES key are erased from memory.

Once the exchange has been completed, the workstation possesses information that it can use to prove the identity of its user for the lifetime of the ticket-granting ticket. As long as the software on the workstation had not been previously tampered with, no information exists that will allow someone else to impersonate the user beyond the life of the ticket.

Requesting a Service

For the moment, let us pretend that the user already has a ticket for the desired server. In order to gain access to the server, the application builds an authenticator containing the client's name and IP address, and the current time. The authenticator is then encrypted in the session key that was received with the ticket for the server. The client then sends the authenticator along with the ticket to the server in a manner defined by the individual application.

Once the authenticator and ticket have been received by the server, the server decrypts the ticket, uses the session key included in the ticket to decrypt the authenticator, compares the information in the ticket with that in the authenticator, the IP address from which the request was received, and the present time. If everything matches, it allows the request to proceed (see Figure 6).

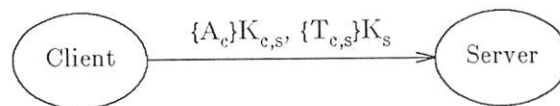


Figure 6. Requesting a Service.

It is assumed that clocks are synchronized to within several minutes. If the time in the request is too far in the future or the past, the server treats the request as an attempt to replay a previous request. The server is also allowed to keep track of all past requests with timestamps that are still valid. In order to further foil replay attacks, a request received with the same ticket and timestamp as one already received can be discarded.

Finally, if the client specifies that it wants the

server to prove its identity too, the server adds one to the timestamp the client sent in the authenticator, encrypts the result in the session key, and sends the result back to the client (see Figure 7).

At the end of this exchange, the server is certain that, according to *Kerberos*, the client is who it says it is. If mutual authentication occurs, the client is also convinced that the server is authentic. Moreover, the client and server share a key which no one else knows, and can safely assume that a reasonably recent message encrypted in that key originated with the other party.

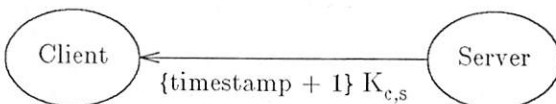


Figure 7. Mutual Authentication.

Getting Server Tickets

Recall that a ticket is only good for a single server. As such, it is necessary to obtain a separate ticket for each service the client wants to use. Tickets for individual servers can be obtained from the ticket-granting service. Since the ticket-granting service is itself a service, it makes use of the service access protocol described in the previous section.

When a program requires a ticket that has not already been requested, it sends a request to the ticket-granting server (see Figure 8). The request contains the name of the server for which a ticket is requested, along with the ticket-granting ticket and an authenticator built as described in the previous section.

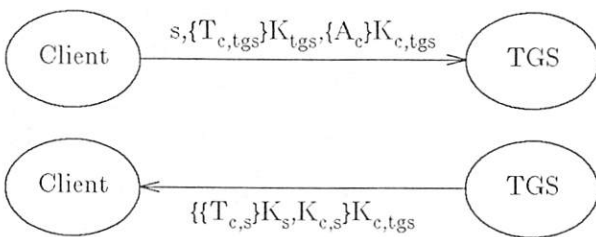


Figure 8. Getting a Server Ticket.

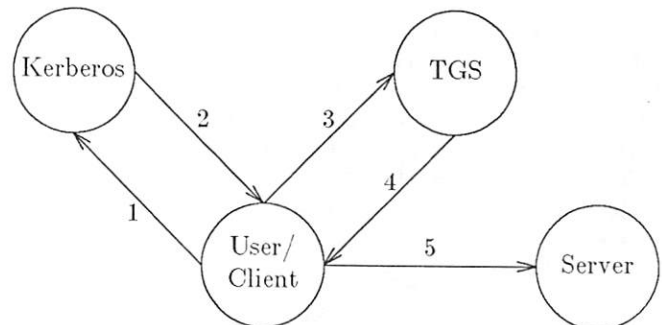
The ticket-granting server then checks the authenticator and ticket-granting ticket as described above. If valid, the ticket-granting server generates a new random session key to be used between the client and the new server. It then builds a ticket for the new server containing the client's name, the server name, the current time, the client's IP address and the new session key it just generated. The lifetime of the new ticket is the minimum of the remaining life for the ticket-granting ticket and the default for the service.

The ticket-granting server then sends the ticket, along with the session key and other information, back to the client. This time, however, the reply is encrypted in the session key that was part of the

ticket-granting ticket. This way, there is no need for the user to enter her/his password again. Figure 9 summarizes the authentication protocols.

The Kerberos Database

Up to this point, we have discussed operations requiring read-only access to the *Kerberos* database. These operations are performed by the authentication service, which can run on both master and slave machines (see Figure 10).



1. Request for TGS ticket
2. Ticket for TGS
3. Request for Server ticket
4. Ticket for Server
5. Request for service

Figure 9. Kerberos Authentication Protocols.

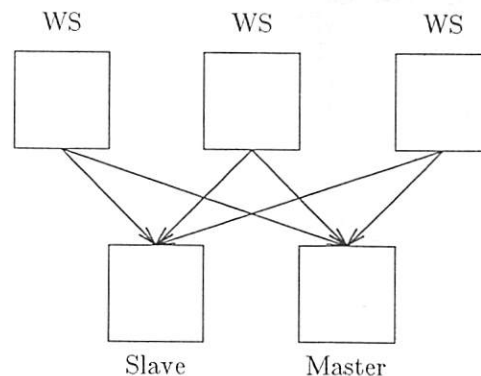


Figure 10. Authentication Requests.

In this section, we discuss operations that require write access to the database. These operations are performed by the administration service, called the *Kerberos* Database Management Service (*KDBM*). The current implementation stipulates that changes may only be made to the master *Kerberos* database; slave copies are read-only. Therefore, the *KDBM* server may only run on the master *Kerberos* machine (see Figure 11). Note that, while authentication can still occur (on slaves), administration requests cannot be serviced if the master machine is down. In our experience, this has not presented a problem, as administration requests are infrequent.

The KDBM handles requests from users to change their passwords. The client side of this program, which sends requests to the KDBM over the network, is the *kpasswd* program. The KDBM also accepts requests from *Kerberos* administrators, who may add principals to the database, as well as change passwords for existing principals. The client side of the administration program, which also sends requests to the KDBM over the network, is the *kadmin* program.

The KDBM Server

The KDBM server accepts requests to add principals to the database or change the passwords for existing principals. This service is unique in that the

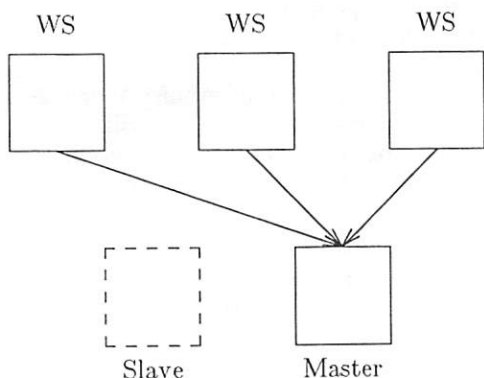


Figure 11. Administration Requests.

ticket-granting service will not issue tickets for it. Instead, the authentication service itself must be used (the same service that is used to get a ticket-granting ticket). The purpose of this is to require the user to enter a password. If this were not so, then if a user left her/his workstation unattended, a passerby could walk up and change her/his password for them, something which should be prevented. Likewise, if an administrator left her/his workstation unguarded, a passerby could change any password in the system.

When the KDBM server receives a request, it authorizes it by comparing the authenticated principal name of the requester of the change to the principal name of the target of the request. If they are the same, the request is permitted. If they are not the same, the KDBM server consults an access control list (stored in a file on the master *Kerberos* system). If the requester's principal name is found in this file, the request is permitted, otherwise it is denied.

By convention, names with a **NULL** instance (the default instance) do not appear in the access control list file; instead, an **admin** instance is used. Therefore, for a user to become an administrator of *Kerberos* an **admin** instance for that username must be created, and added to the access control list. This convention allows an administrator to use a different password for *Kerberos* administration than s/he would use for normal login.

All requests to the KDBM program, whether

permitted or denied, are logged.

The *kadmin* and *kpasswd* Programs

Administrators of *Kerberos* use the *kadmin* program to add principals to the database, or change the passwords of existing principals. An administrator is required to enter the password for their *admin* instance name when they invoke the *kadmin* program. This password is used to fetch a ticket for the KDBM server (see Figure 12).

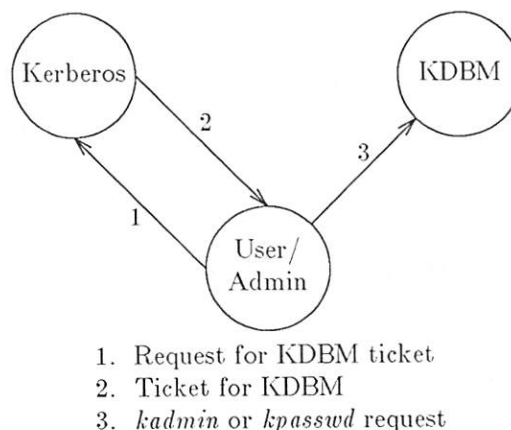


Figure 12. Kerberos Administration Protocol.

Users may change their *Kerberos* passwords using the *kpasswd* program. They are required to enter their old password when they invoke the program. This password is used to fetch a ticket for the KDBM server.

Database Replication

Each *Kerberos* realm has a *master Kerberos* machine, which houses the master copy of the authentication database. It is possible (although not necessary) to have additional, read-only copies of the database on *slave* machines elsewhere in the system. The advantages of having multiple copies of the database are those usually cited for replication: higher availability and better performance. If the master machine is down, authentication can still be achieved on one of the slave machines. The ability to perform authentication on any one of several machines reduces the probability of a bottleneck at the master machine.

Keeping multiple copies of the database introduces the problem of data consistency. We have found that very simple methods suffice for dealing with inconsistency. The master database is dumped every hour. The database is sent, in its entirety, to the slave machines, which then update their own databases. A program on the master host, called *kprop*, sends the update to a peer program, called *kproxd*, running on each of the slave machines (see Figure 13). First *kprop* sends a checksum of the new database it is about to send. The checksum is encrypted in the *Kerberos* master database key, which both the master and slave *Kerberos* machines possess. The data is then transferred over the network to the

kpropd on the slave machine. The slave propagation server calculates a checksum of the data it has received, and if it matches the checksum sent by the master, the new information is used to update the slave's database.

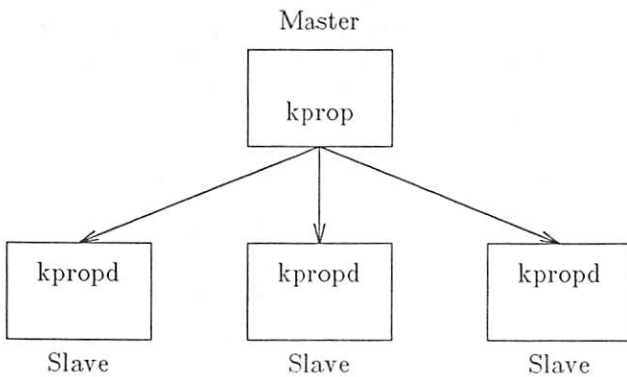


Figure 13. Database Propagation.

All passwords in the *Kerberos* database are encrypted in the master database key. Therefore, the information passed from master to slave over the network is not useful to an eavesdropper. However, it is essential that only information from the master host be accepted by the slaves, and that tampering of data be detected, thus the checksum.

Kerberos From the Outside Looking In

The section will describe *Kerberos* from the practical point of view, first as seen by the user, then from the application programmer's viewpoint, and finally, through the tasks of the *Kerberos* administrator.

User's Eye View

If all goes well, the user will hardly notice that *Kerberos* is present. In our UNIX implementation, the ticket-granting ticket is obtained from *Kerberos* as part of the *login* process. The changing of a user's *Kerberos* password is part of the *passwd* program. And *Kerberos* tickets are automatically destroyed when a user logs out.

If the user's login session lasts longer than the lifetime of the ticket-granting ticket (currently 8 hours), the user will notice *Kerberos*' presence because the next time a *Kerberos*-authenticated application is executed, it will fail. The *Kerberos* ticket for it will have expired. At that point, the user can run the *kinit* program to obtain a new ticket for the ticket-granting server. As when logging in, a password must be provided in order to get it. A user executing the *klist* command out of curiosity may be surprised at all the tickets which have silently been obtained on her/his behalf for services which require *Kerberos* authentication.

From the Programmer's Viewpoint

A programmer writing a *Kerberos* application will often be adding authentication to an already existing network application consisting of a client and server side. We call this process "Kerberizing" a program. Kerberizing usually involves making a call to the *Kerberos* library in order to perform authentication at the initial request for service. It may also involve calls to the DES library to encrypt messages and data which are subsequently sent between application client and application server.

The most commonly used library functions are *krb_mk_req* on the client side, and *krb_rd_req* on the server side. The *krb_mk_req* routine takes as parameters the name, instance, and realm of the target server, which will be requested, and possibly a checksum of the data to be sent. The client then sends the message returned by the *krb_mk_req* call over the network to the server side of the application. When the server receives this message, it makes a call to the library routine *krb_rd_req*. The routine returns a judgement about the authenticity of the sender's alleged identity.

If the application requires that messages sent between client and server be secret, then library calls can be made to *krb_mk_priv* (*krb_rd_priv*) to encrypt (decrypt) messages in the session key which both sides now share.⁷

The *Kerberos* Administrator's Job

The *Kerberos* administrator's job begins with running a program to initialize the database. Another program must be run to register essential principals in the database, such as the *Kerberos* administrator's name with an **admin** instance. The *Kerberos* authentication server and the administration server must be started up. If there are slave databases, the administrator must arrange that the programs to propagate database updates from master to slaves be kicked off periodically.

After these initial steps have been taken, the administrator manipulates the database over the network, using the *kadmin* program. Through that program, new principals can be added, and passwords can be changed.

In particular, when a new *Kerberos* application is added to the system, the *Kerberos* administrator must take a few steps to get it working. The server must be registered in the database, and assigned a private key (usually this is an automatically generated random key). Then, some data (including the server's key) must be extracted from the database and installed in a file on the server's machine. The default file is */etc/srvtab*. The *krb_rd_req* library routine called by the server (see the previous section) uses the information in that file to decrypt messages sent encrypted in the server's private key. The */etc/srvtab* file authenticates the server as a password typed at a terminal authenticates the user.

The *Kerberos* administrator must also ensure

that *Kerberos* machines are physically secure, and would also be wise to maintain backups of the Master database.⁸

The Bigger Picture

In this section, we describe how *Kerberos* fits into the Athena environment, including its use by other network services and applications, and how it interacts with remote *Kerberos* realms. For a more complete description of the Athena environment, please see G. W. Treese.⁹

Other Network Services' Use of *Kerberos*

Several network applications have been modified to use *Kerberos*. The *rlogin* and *rsh* commands first try to authenticate using *Kerberos*. A user with valid *Kerberos* tickets can *rlogin* to another Athena machine without having to set up *.rhosts* files. If the *Kerberos* authentication fails, the programs fall back on their usual methods of authorization, in this case, the *.rhosts* files.

We have modified the Post Office Protocol to use *Kerberos* for authenticating users who wish to retrieve their electronic mail from the "post office". A message delivery program, called *Zephyr*, has been recently developed at Athena, and it uses *Kerberos* for authentication as well.¹⁰

The program for signing up new users, called *register*, uses both the Service Management System (SMS)¹¹ and *Kerberos*. From SMS, it determines whether the information entered by the would-be new Athena user, such as name and MIT identification number, is valid. It then checks with *Kerberos* to see if the requested username is unique. If all goes well, a new entry is made to the *Kerberos* database, containing the username and password.

For a detailed discussion of the use of *Kerberos* to secure Sun's Network File System, please refer to the appendix.

Interaction with Other Kerberis

It is expected that different administrative organizations will want to use *Kerberos* for user authentication. It is also expected that in many cases, users in one organization will want to use services in another. *Kerberos* supports multiple administrative domains. The specification of names in *Kerberos* includes a field called the *realm*. This field contains the name of the administrative domain within which the user is to be authenticated.

Services are usually registered in a single realm and will only accept credentials issued by an authentication server for that realm. A user is usually registered in a single realm (the local realm), but it is possible for her/him to obtain credentials issued by another realm (the remote realm), on the strength of the authentication provided by the local realm. Credentials valid in a remote realm indicate the realm in which the user was originally authenticated. Services in the remote realm can choose whether to honor

those credentials, depending on the degree of security required and the level of trust in the realm that initially authenticated the user.

In order to perform cross-realm authentication, it is necessary that the administrators of each pair of realms select a key to be shared between their realms. A user in the local realm can then request a ticket-granting ticket from the local authentication server for the ticket-granting server in the remote realm. When that ticket is used, the remote ticket-granting server recognizes that the request is not from its own realm, and it uses the previously exchanged key to decrypt the ticket-granting ticket. It then issues a ticket as it normally would, except that the realm field for the client contains the name of the realm in which the client was originally authenticated.

This approach could be extended to allow one to authenticate oneself through a series of realms until reaching the realm with the desired service. In order to do this, though, it would be necessary to record the entire path that was taken, and not just the name of the initial realm in which the user was authenticated. In such a situation, all that is known by the server is that A says that B says that C says that the user is so-and-so. This statement can only be trusted if everyone along the path is also trusted.

Issues and Open Problems

There are a number of issues and open problems associated with the *Kerberos* authentication mechanism. Among the issues are how to decide the correct lifetime for a ticket, how to allow proxies, and how to guarantee workstation integrity.

The ticket lifetime problem is a matter of choosing the proper tradeoff between security and convenience. If the life of a ticket is long, then if a ticket and its associated session key are stolen or misplaced, they can be used for a longer period of time. Such information can be stolen if a user forgets to log out of a public workstation. Alternatively, if a user has been authenticated on a system that allows multiple users, another user with access to root might be able to find the information needed to use stolen tickets. The problem with giving a ticket a short lifetime, however, is that when it expires, the user will have to obtain a new one which requires the user to enter the password again.

An open problem is the proxy problem. How can an authenticated user allow a server to acquire other network services on her/his behalf? An example where this would be important is the use of a service that will gain access to protected files directly from a fileserver. Another example of this problem is what we call *authentication forwarding*. If a user is logged into a workstation and logs in to a remote host, it would be nice if the user had access to the same services available locally, while running a program on the remote host. What makes this difficult is that the user might not trust the remote host, thus

authentication forwarding is not desirable in all cases. We do not presently have a solution to this problem.

Another problem, and one that is important in the Athena environment, is how to guarantee the integrity of the software running on a workstation. This is not so much of a problem on private workstations since the user that will be using it has control over it. On public workstations, however, someone might have come along and modified the *login* program to save the user's password. The only solution presently available in our environment is to make it difficult for people to modify software running on the public workstations. A better solution would require that the user's key never leave a system that the user knows can be trusted. One way this could be done would be if the user possessed a *smartcard* capable of doing the encryptions required in the authentication protocol.

Status

A prototype version of *Kerberos* went into production in September of 1986. Since January of 1987, *Kerberos* has been Project Athena's sole means of authenticating its 5,000 users, 650 workstations, and 65 servers. In addition, *Kerberos* is now being used in place of *.rhosts* files for controlling access in several of Athena's timesharing systems.

Acknowledgements

Kerberos was initially designed by Steve Miller and Clifford Neuman with suggestions from Jeff Schiller and Jerry Saltzer. Since that time, numerous other people have been involved with the project. Among them are Jim Aspnes, Bob Baldwin, John Barba, Richard Basch, Jim Bloom, Bill Bryant, Mark Colan, Rob French, Dan Geer, John Kohl, John Kubiatowicz, Bob McKie, Brian Murphy, John Ostlund, Ken Raeburn, Chris Reed, Jon Rochlis, Mike Shanzer, Bill Sommerfeld, Ted T'so, Win Treese, and Stan Zanarotti.

We are grateful to Joshua Lubarr, Dan Geer, Kathy Lieben, Ken Raeburn, Jerry Saltzer, Ed Steiner, Robbert van Renesse, and Win Treese whose suggestions much improved earlier drafts of this paper.

The illustration of the three-headed dog is by Besty Bruemmer.

Appendix

Kerberos Application to SUN's Network File System (NFS)

A key component of the Project Athena workstation system is the interposing of the network between the user's workstation and her/his private file storage (home directory). All private storage resides on a set of computers (currently VAX 11/750s) that are dedicated to this purpose. This allows us to offer services on publicly available UNIX workstations. When a user logs in to one of these publicly

available workstations, rather than validate her/his name and password against a locally resident password file, we use *Kerberos* to determine her/his authenticity. The *login* program prompts for a username (as on any UNIX system). This username is used to fetch a *Kerberos* ticket-granting ticket. The *login* program uses the password to generate a DES key for decrypting the ticket. If decryption is successful, the user's home directory is located by consulting the *Hesiod* naming service and mounted through NFS. The *login* program then turns control over to the user's shell, which then can run the traditional per-user customization files because the home directory is now "attached" to the workstation. The *Hesiod* service is also used to construct an entry in the local password file. (This is for the benefit of programs that look up information in */etc/passwd*.)

From several options for delivery of remote file service, we chose SUN's Network File System. However this system fails to mesh with our needs in a crucial way. NFS assumes that all workstations fall into two categories (as viewed from a file server's point of view): trusted and untrusted. Untrusted systems cannot access any files at all, trusted can. Trusted systems are completely trusted. It is assumed that a trusted system is managed by friendly management. Specifically, it is possible from a trusted workstation to masquerade as any valid user of the file service system and thus gain access to just about every file on the system. (Only files owned by "root" are exempted.)

In our environment, the management of a workstation (in the traditional sense of UNIX system management) is in the hands of the user currently using it. We make no secret of the root password on our workstations, as we realize that a truly unfriendly user can break in by the very fact that s/he is sitting in the same physical location as the machine and has access to all console functions. Therefore we cannot truly trust our workstations in the NFS interpretation of trust. To allow proper access controls in our environment we had to make some modifications to the base NFS software, and integrate *Kerberos* into the scheme.

Unmodified NFS

In the implementation of NFS that we started with (from the University of Wisconsin), authentication was provided in the form of a piece of data included in each NFS request (called a "credential" in NFS terminology). This credential contains information about the unique user identifier (UID) of the requester and a list of the group identifiers (GIDs) of the requester's membership. This information is then used by the NFS server for access checking. The difference between a trusted and a non-trusted workstation is whether or not its credentials are accepted by the NFS server.¹²

Modified NFS

In our environment, NFS servers must accept credentials from a workstation if and only if the credentials indicate the UID of the workstation's user, and no other.

One obvious solution would be to change the nature of credentials from mere indicators of UID and GIDs to full blown *Kerberos* authenticated data. However a significant performance penalty would be paid if this solution were adopted. Credentials are exchanged on every NFS operation including all disk read and write activities. Including a *Kerberos* authentication on each disk transaction would add a fair number of full-blown encryptions (done in software) per transaction and, according to our envelope calculations, would have delivered unacceptable performance. (It would also have required placing the *Kerberos* library routines in the kernel address space.)

We needed a hybrid approach, described below. The basic idea is to have the NFS server map credentials received from client workstations, to a valid (and possibly different) credential on the server system. This mapping is performed in the server's kernel on each NFS transaction and is setup at "mount" time by a user-level process that engages in *Kerberos*-moderated authentication prior to establishing a valid kernel credential mapping.

To implement this we added a new system call to the kernel (required only on server systems, not on client systems) that provides for the control of the mapping function that maps incoming credentials from client workstations to credentials valid for use on the server (if any). The basic mapping function maps the tuple:

`<CLIENT-IP-ADDRESS, UID-ON-CLIENT>`

to a valid NFS credential on the server system. The CLIENT-IP-ADDRESS is extracted from the NFS request packet and the UID-ON-CLIENT is extracted from the credential supplied by the client system. Note: all information in the client-generated credential except the UID-ON-CLIENT is discarded.

If no mapping exists, the server reacts in one of two ways, depending it is configured. In our friendly configuration we default the unmappable requests into the credentials for the user "nobody" who has no privileged access and has a unique UID. Unfriendly servers return an NFS access error when no valid mapping can be found for an incoming NFS credential.

Our new system call is used to add and delete entries from the kernel resident map. It also provides the ability to flush all entries that map to a specific UID on the server system, or flush all entries from a given CLIENT-IP-ADDRESS.

We modified the mount daemon (which handles NFS mount requests on server systems) to accept a new transaction type, the *Kerberos* authentication

mapping request. Basically, as part of the mounting process, the client system provides a *Kerberos* authenticator along with an indication of her/his UID-ON-CLIENT (encrypted in the *Kerberos* authenticator) on the workstation. The server's mount daemon converts the *Kerberos* principal name into a local username. This username is then looked up in a special file to yield the user's UID and GIDs list. For efficiency, this file is a *ndbm* database file with the username as the key. From this information, an NFS credential is constructed and handed to the kernel as the valid mapping of the `<CLIENT-IP-ADDRESS, CLIENT-UID>` tuple for this request.

At unmount time a request is sent to the mount daemon to remove the previously added mapping from the kernel. It is also possible to send a request at logout time to invalidate all mapping for the current user on the server in question, thus cleaning up any remaining mappings that exist (though they shouldn't) before the workstation is made available for the next user.

Security Implications of the Modified NFS

This implementation is not completely secure. For starters, user data is still sent across the network in an unencrypted, and therefore interceptable, form. The low-level, per-transaction authentication is based on a `<CLIENT-IP-ADDRESS, CLIENT-UID>` pair provided unencrypted in the request packet. This information could be forged and thus security compromised. However, it should be noted that only while a user is actively using her/his files (i.e., while logged in) are valid mappings in place and therefore this form of attack is limited to when the user in question is logged in. When a user is not logged in, no amount of IP address forgery will permit unauthorized access to her/his files.

References

1. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Section E.2.1: Kerberos Authentication and Authorization System*, M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).
2. E. Balkovich, S. R. Lerman, and R. P. Parmelee, "Computing in Higher Education: The Athena Experience," *Communications of the ACM* **28**(11) pp. 1214-1224 ACM, (November, 1985).
3. R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM* **21**(12) pp. 993-999 (December, 1978).
4. V. L. Voydock and S. T. Kent, "Security Mechanisms in High-Level Network Protocols," *Computing Surveys* **15**(2)ACM, (June 1983).
5. National Bureau of Standards, "Data Encryption Standard," Federal Information Processing

Standards Publication 46, Government Printing Office, Washington, D.C. (1977).

6. S. P. Dyer, "The Hesiod Name Server," in *Usenix Conference Proceedings*, (Winter, 1988).
7. W. J. Bryant, *Kerberos Programmer's Tutorial*, M.I.T. Project Athena (In preparation).
8. W. J. Bryant, *Kerberos Administrator's Manual*, M.I.T. Project Athena (In preparation).
9. G. W. Treese, "Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD," in *Usenix Conference Proceedings*, (Winter, 1988).
10. C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, "The Zephyr Notification System," in *Usenix Conference Proceedings*, (Winter, 1988).
11. M. A. Rosenstein, D. E. Geer, and P. J. Levine, "Take Me To Your Leader: Service Management in a Complicated Environment," in *Usenix Conference Proceedings*, (Winter, 1988).
12. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," in *Usenix Conference Proceedings*, (Summer, 1985).



The Athena Service Management System

Mark A. Rosenstein
Daniel E. Geer, Jr.
Peter J. Levine
Project Athena
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
{mar,geer,pjlevine}@ATHENA.MIT.EDU

ABSTRACT

Maintaining, managing, and supporting an unbounded number of distributed network services on multiple server instances requires new solutions. The Athena Service Management System provides centralized control of data administration, a protocol for interface to the database, tools for accessing and modifying the database, and an automated mechanism for data distribution.

Introduction

The purpose of the Athena Service Management System (SMS) is to provide a single point of contact for authoritative information about resources and services in a distributed computing environment. SMS is a centralized data administrator providing network-based update and maintenance of system servers.[†]

- Conceptually, SMS provides mechanisms for managing servers and resources. This aspect comprises the fundamental design of SMS.
- Economically, SMS provides a replacement for labor-intensive hand-management of server configuration files.
- Technically, SMS consists of a database, a server and its protocol interface, a data distribution manager, and tools for accessing and modifying SMS data.

SMS provides coherent data access and data update. Access to data is provided through a standard application interface. Programs designed to reconfigure network servers, edit mailing lists, manage group membership, etc., all use this application interface. Applications used as administrative tools are invoked by users; applications that update servers are (automatically) invoked at regular intervals. Management reports and operational feedback are also provided.

[†] Care must be taken to distinguish among different senses of the word "server": the SMS server, which manages a database; the servers which provide system services and whose maintenance are SMS's reason for existence; and the update servers which allow SMS to affect these system services.

Requirements

The requirements for the initial Athena SMS include:

- Management of 15,000 accounts, including individual users, course, and project accounts, and special accounts used by system services.
- Management of 1,000 workstations, timesharing machines, and network servers, including specification of default resource assignments.
- Allocation of controlled network services, such as creating and setting quotas for new users' home directories on network file servers, consistent with load-balancing constraints.
- Maintenance of other control information, including user groups, mailing lists, access control lists, etc.
- Maintenance of resource directories, such as the location of printers, specialized file systems (including privately supported file systems), and other network services.

This must be accomplished with the utmost robustness.

The system must be easily expandable, both to support additional instances of a particular service and to offer additional services in the future. At this time, SMS is used to update three services (with RVD to be added shortly):

- *Hesiod*: The Athena Name Service, *Hesiod*, provides service-to-server and label translation. It can be thought of as a high-performance, read-only front-end to the SMS database. See the companion paper in this volume.¹
- NFS: At Athena, most shared-access read-write file systems are provided by a locally modified form of the Network File System.² SMS manages the NFS

server hosts, providing quota-based resource allocation, and load balancing. Also refer to the companion paper in this volume.³

- Mail Service: Athena's mail service is through a central routing hub to multiple post office servers (mail repositories) based on the Post Office Protocol⁴ (POP) of the Rand Mail Handler⁵ (MH) package. SMS allocates individual post office boxes to post office servers, and builds the `/usr/lib/aliases` control file used on the central mail hub.
- RVD: At Athena, most shared-access read-only file systems are provided by a Remote Virtual Disk system. SMS manages the RVD server hosts, providing access control lists and server configuration files.

Each of these server hosts are controlled by some number of server-specific data files; over 50 separate files are required to support the services described above. SMS currently supports three *Hesiod* servers, 17 RVD file servers, 32 NFS file servers, one mail hub and three post offices. Each RVD server requires one file, and each server's file is different. A *Hesiod* server requires 9 separate files containing 64000 resolvable queries, but each *Hesiod* server receives the same 9 files. Each NFS server requires two files, one file identical across most NFS servers, one file different. The mail hub requires one file, `/usr/lib/aliases`.

Design Points

There are five factors to be taken into account when making design decisions. In order of importance, to this project they are:

1. Reliability
2. Consistency
3. Flexibility
4. Time Efficiency
5. Space Efficiency

SMS must be reliable. In particular, it must be designed to allow straightforward recreation of SMS on replacement hardware from backups, should the need arise. The backup regime for SMS consists of frequent database backups in ASCII format, to redundant sites. All components of the SMS system are designed such that a duplicate configuration can be kept running as a test platform without interfering with normal, "real," operations. Service updates are verified by testing the server before calling an update successful. Failed operations ring alarms that are heard, disabling parts of the system if required, but without interactions with logically unconnected parts of the service management system. Services which are duplicated for availability are updated such that the service is always available; i.e., it is not permitted for server updates to drift into unwarranted synchrony, bringing down all instances at the same time. The entire life-cycle is considered part of the package, so tight change and source-code control (reviewing each change to source, running only what can be built from source) is part of the design.

In addition to having authoritative control of the data, SMS must see that the data is kept consistent. To guarantee internal consistency, SMS clients do not touch the database directly; they do not even see the database system used by SMS. Each application uses the application library to access the database. This library is a collection of functions allowing access to the database by communicating with the SMS server using the SMS protocol. Many of the database consistency constraints are handled by the library. A number of consistency verification applications also exist. To make this consistency reliable, the protocol is designed to be tamper-proof, withstanding both denial-of-service attacks and malicious attempts to corrupt the data. Security is provided with the help of authentication by the *Kerberos* private-key authentication system. See the companion paper in this volume.³

Beyond these goals, SMS must be flexible in both its database underpinnings and the services it supports. As discussed later in the design section, the particular Database Management System used is insulated from SMS through a Global DataBase (GDB) library,⁶ making SMS plug-compatible with other database foundations. It is independent of the individual services—while each service updated by SMS has its own particular data format and structure, the SMS database stores data in one coherent format. A separate program, the Data Control Manager (DCM), converts SMS database (internal) structure to server-appropriate structure (such as a configuration file). When a new service is supported, the database can be changed and only minimal updates are necessary in the SMS server, and a new module is added to the DCM specifying the additional specific output data format to be manufactured.

Also, in the interests of flexibility, no administrative policy decisions are coded into the design. These are determined only by the data in the database.

Simplicity of the design is more important than the speed of operation; other systems, such as the *Hesiod* name service, provide a high bandwidth read-only interface to the database. To this end, the server only supports simple queries. Processing efficiency for complex queries is maximized by local applications running on the workstation, not on the central server. Any set of changes that must be atomic to maintain database consistency are performed on the server; sets of non-atomic changes and complex lookup operations are supported in the server only through combining simpler queries.

System Design

There are three sides to the SMS system:

- The input side, which contains all of the user-interface programs, allowing the user to enter, examine, or modify data in the database.
- The database side, which consists of the actual database, the SMS server which manipulates the

database, and utility programs to backup, restore, and verify the internal consistency of the database.

- The output side, which extracts information from the database, formats it into server-specific configuration files, and updates the various servers by propagating these files.



Various amounts of glue are required to connect these three sides. At the lowest level, there is the network protocol that client programs use to gain access to the database. This, however, depends on the actual model of how database queries are done, which is influenced by the organization of the database (but not its exact format, which is hidden through the protocol).

The SMS Protocol

The SMS protocol is the fundamental interface to SMS for client applications. It allows all clients of SMS to speak a common, network-transparent language.

The SMS protocol is a remote procedure call protocol based on the GDB library, which in turn uses a TCP stream. Each client program makes a connection to a well known port to contact the SMS server, sending requests and receiving replies over that stream. Each request consists of a major request number, and several counted strings of bytes. Each reply consists of a single status code followed by zero or more counted strings of bytes. Requests and replies also contain a protocol version number, to allow clean handling of version skew.

The following major requests are defined for SMS. It should be noted that each query defines its own signature of arguments and results. For some of these actions the server checks authorization based on the authenticated identity of the user making the request.

noop Do nothing. This is useful for testing and profiling of the server.

authenticate There is one argument, a *Kerberos*⁷ authenticator. All requests received after this request are performed on behalf of the principal identified by the authenticator.

query The first argument is the name of a pre-defined query (a "query handle"), and the rest are arguments to that query. Queries may retrieve information or modify what is in the database. If the server query is allowed, any retrieved data are passed back as several return values. All but the last returned value will have a status code indicating more data, with the final one returning the real status code of the

query.

access There are a variable number of arguments. The first is the name of a pre-defined query usable for the "query" request, and the rest are query arguments. The server returns a reply with a zero status code if the query would have been allowed, or a reply with a non-zero status code explaining why the query was rejected.

Normal use of the protocol consists of establishing a connection, providing authentication, then performing a series of queries. As long as the application only wants to retrieve data or perform simple updates, only an authenticate followed by queries are necessary. The access operation is useful for verifying that an operation with side effects will succeed before attempting it.

Queries

All access to the database by clients is provided by the application library via the SMS protocol. This interface provides a limited set of predefined, named queries, allowing tightly controlled access to database information. Queries fall into four classes: retrieve, update, delete, and append. An attempt has been made to define a set of queries that provide sufficient flexibility to meet all of the needs of the Data Control Manager as well as the individual application programs, since the DCM uses the same interface as the clients to read the database. Since the database can be modified and extended, the application library has been designed to allow the easy addition of queries.

The generalized layer of functions makes SMS independent of the underlying database. If a different database management system is required, the only change needed will be a new SMS server. It is made by linking the pre-defined queries to a set of data manipulation procedures provided by a version of GDB suited to the alternate DBMS.

At this time there are over 100 supported queries. See the complete technical description of SMS for a listing.⁸ Some sample queries include:

get_nfs_quotas

Arguments: machine, device

Returns: list of login/quota tuples

Errors: no match, bad machine

Retrieves disk quota assignments for all users on the specified disk partition.

get_user_by_login

Arguments: login name

Returns: login, uid, shell, home, last, first, middle, status, ID number, year, expiration date, modification time

Errors: no match, not unique

Retrieves information about a particular user, searching by login name. Similar queries exist to search by last name, first name, user ID, and Registrar's ID number.

add_machine

Arguments: name, type, model, status, serial

number, system type
 Returns: nothing
 Errors: already exists, bad type
Appends a new machine to the list of machines known by SMS.

update_server_info

Arguments: service, update interval, target file, script, dfgen
 Returns: nothing
 Errors: no match, not unique
Updates a service entry, allowing anything but the service name to be changed.

delete_filesys

Arguments: label
 Returns: nothing
 Errors: no match, not unique
Deletes the specified file system information from the database. Does not automatically reclaim the storage at this time.

Each query has two possible return status values in addition to any errors given above: **success** and **permission_denied**.

The Database Management System

The database is the core of SMS. It provides the storage mechanism for the complete system. SMS expects its database to consist of several tables of records with strings, integers, and dates. Tables are keyed on one or more fields in each record allowing efficient retrieval by key or wild cards.

The database currently in use at Athena is Ingres⁹ from Relational Technology, Inc. Ingres provides a complete query system, a C library of routines, and a command interface language. Its advantages are that it is available and that it mostly works. By design, SMS does not depend on any special feature of Ingres so as to retain the option to utilize other relational database systems.

The database is an independent entity from the SMS system. The Ingres query bindings and database-specific routines are layered at the lower levels of the SMS server. All applications are independent of database-specific routines. An application passes query handles to the SMS server which then resolves the request into an appropriate database query.

The database contains several types of objects. Each object in the database has an access control list (ACL) associated with it indicating who is allowed to modify that object. Each object also has records who last modified it and when that modification was performed. The ACL's are just references to lists in the database. The database contains the following:

- User information: full name, login name, user ID, registrar's ID, login shell, home directory, class year, status, modification time, nickname, home address, home phone, office address, office phone, school affiliation, an ACL
- Machine information: name, type, model, status,

serial number, system type, an ACL

- Cluster (mapping of machines to default printer and RVD server) information: name, description, location, default servers, machine assignments, an ACL
- General service information: service name to network port mapping
- File system configuration: name, type, server host, name on server host, mount point on client host, access mode, an ACL
- NFS information: host name, physical disk partition, quota by user on each partition, an ACL
- RVD information: host name, physical disk partition, virtual disks assigned to each partition, pack ID, access passwords, size, an ACL
- Printer information: name to */etc/printcap* entry mapping
- Post office location: for each user post office server and box on that server
- Lists: name, description, modification date, members (which can be users, other lists, or literal strings), attributes (mailing list, UNIX group and gid), an ACL
- Aliases: includes allowed keywords for certain fields in the database, alternate names for file systems, alternate names for services
- DCM information: services to be updated, hosts supporting each service, target files on each host, last update time, enable/override/success flags for updates
- Internal control information: next user ID, group ID, machine ID, and cluster ID to assign (these are just hints); an ACL for each query; usage statistics

SMS-to-Server Update Protocol

SMS provides a reliable mechanism for updating the servers it manages. The goals of the server update protocol are:

- Rational, automatic update for normal cases and expected kinds of failures.
- Ability to survive clean (target) server crashes.
- Ability to survive clean SMS crashes.
- Easily understood state so that straightforward recovery by hand is possible.

All actions are initiated by the SMS system. Updates of managed servers are performed such that a partially completed update is harmless. Updates not completed are simply rescheduled for retry until they succeed, or until an update can not be initiated. In the latter case, a human operator will be notified.

The update protocol is based on the GDB library just as is the SMS protocol itself. In the update protocol, there are four commands:

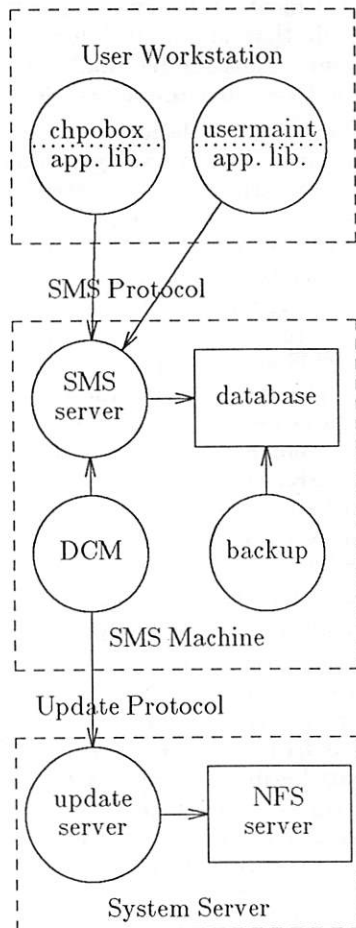
authenticate A Kerberos authenticator is sent. The update server on the target server uses this authenticator to verify that the entity

contacting it is authorized to initiate updates.

transfer This command is used for sending information, usually entire files. The protocol is capable of efficiently sending a half-megabyte binary file.

instructions This sends over a command script, which when executed on the target server will install the new configuration files just sent.

execute This instructs the update server to execute the instructions just sent.



In typical usage, all four commands are used in the order presented above. Multiple transfers may be necessary for some server types. Note that when data files are transferred, they do not directly overwrite the existing data files. Instead, they are put in a temporary position. When the update is executed, the old files are renamed to another temporary name, and the new files are given the correct names. This minimizes the interval during which a server system crash would leave an inconsistent set of configuration files. If all portions of the preparation are completed without error, the execution is then allowed. This usually consists of moving files around, then sending a signal to or restarting a server process. The update server then performs a plausibility check on the result by verifying the operation of the system server, and sends SMS

an indicative return code.

System Components

Six software components make up the SMS system.

- The database, currently built on the commercial database system RTI Ingres.
- The SMS server, a program always running on the machine containing the database. It accesses the database for the client programs.
- The application library, a collection of routines implementing the SMS protocol. It is used by client programs to communicate with the SMS server.
- The client programs, a collection of programs that make up the user interface to the system.
- The data control manager, a program run periodically by *cron* and driven by data in the database. It constructs up-to-date server configuration files and installs these files on the servers.
- The update servers, which run on each machine containing a server that SMS updates. These are contacted by the data control manager to install the new configuration files and notify the real servers being managed by SMS.

Because SMS has a variety of interfaces, a distinction must be maintained between applications called clients that directly read and write to SMS (i.e., administrative programs) and services which use information distributed from SMS (i.e., a name server). In both cases the interface to the SMS database is through the SMS server, using the SMS protocol. The significant difference is that server update is handled automatically through the Data Control Manager; administrative programs are executed by users.

Clients

SMS includes a set of specialized management tools to grant system administrators overall control of system resources. For each system service there is an administrative interface. To accommodate novice and occasional users, a menu interface is the default. For regular users, a command-line switch is provided that will use a line-oriented interface. This provides speed and directness for users familiar with the system, while being reasonably helpful to novices and occasional users. A specialized menu building tool has been developed in order that new application programs can be developed quickly. This user interface does not depend on the X Window System.¹⁰ It must be possible for system operators to use dumb terminals to correct resource problems, i.e. it cannot be a requirement that a high level of functionality be present before the service management system can operate.

Fields in the database have associated with them lists of legal values. A null list indicates that any value is possible. This is useful for fields such as *user_name*, *address*, and so forth. The application

programs, before attempting to modify anything in the database, request this information, and compare it with the proposed new value. If an invalid value is discovered, it is reported to the user, who is given the opportunity to change the value, or "insist" that it is a new, legal value. (The ability to update data in the database does not necessarily indicate the ability to add new legal values to the database.)

Applications should be aware of the ramifications of their actions, and notify the user if appropriate. For example, an administrator deleting a user is informed of storage space that is being reclaimed, mailing lists that are being modified, and so forth. Objects that need to be modified at once (such as the ownership of a mailing list) present themselves to be dealt with.

The following list of client programs are currently in use at Project Athena. These are rewrites of standard UNIX programs, and are available to regular users:

- *chfn* - change finger information
- *chsh* - change default shell

These are new programs available to regular users:

- *register* - allow new students to claim an Athena account
- *mailmaint* - allow users to add/delete themselves to mailing lists

These are used by system administrators:

- *attachmaint* - map file system names to physical server configurations
- *chpobox* - change forwarding post office for a user
- *clustermaint* - associate a machine with a set of default servers
- *dcmaint* - update DCM table entries, including service/machine mapping
- *listmaint* - create and maintain groups, mailing lists, and access control lists
- *nfsmaint* - configure NFS file servers
- *portmaint* - maintain the list of well known contact ports
- *regtape* - enter new students from the Registrar's tape
- *rvdmaint* - configure RVD servers
- *usermaint* - maintain user information including file service and post office location

Finally, this program is used only in debugging SMS:

- *smstest* - perform any query manually

New User Registration

A specialized client is the new user registration program. A new student must be able to claim an Athena account without any intervention from Athena user accounts staff. Without this the user accounts staff would be faced with manually creating hundreds of accounts at the beginning of each academic term.

Athena obtains a copy of the official list of registered students from the MIT Registrar shortly before the start of each term. The *regtape* client adds each student on the Registrar's tape who has not already been registered for an Athena account to the "users" relation of the database, and assigned a unique user ID; the student is not assigned a login name or any other resources at this time. A one-way encrypted form of the student's ID number is stored along with the name. No other database resources are allocated at that time. This ID number and the exact spelling of the student's full name as recorded by the Registrar are all that are needed for a student to claim an account. Thus the ID number is something of a password until a real account has been set up.

When the student decides to register with Athena, he or she walks up to any workstation and logs in using the username of "register". This produces a form-like interface prompting for the user's first name, middle initial, last name, and student ID number. The *register* program does not talk to the SMS server directly, but goes through a registration server first. The registration server deals with access control to the SMS server and the *Kerberos* administration server. Register encrypts the ID number, and sends a *verify_user* request to the registration server. The server responds with **already_registered**, **not_found**, or **OK**. After this the register server will do work on behalf of the user; the user still cannot contact SMS directly until he or she obtains a login name and user ID.

If the user has been validated, *register* then prompts the student for a choice in login names. It then goes through a two-step process to verify the login name: first, it simulates a login request for this user name with *Kerberos*; if this fails (indicating that the username is free and may be registered), it then sends a **grab_login** request. On receiving a **grab_login** request, the registration server then proceeds to register the login name with *Kerberos*; if the login name is already in use, it returns a failure code to *register*. Otherwise, it allocates a home directory for the user on the least-loaded fileserver, sets an initial disk quota for the user, builds a post office entry for the user on the least-loaded post office server, and returns a success code to *register*. SMS keeps track of loading on servers in the database, incrementing and decrementing its estimate of the load as it allocates and deallocates resources on each server.

Register then prompts the user for an initial password. It sends a **set_password** request to the registration server, which decrypts it and forwards it to *Kerberos*. At this point, pending propagation of information to the *Hesiod* name service, the central mail hub, and the user's home fileserver, the user's account has been established. These updates may take up to 12 hours to complete.

The SMS Server

As previously stated, all remote communication with the SMS database is done through the SMS server, using the SMS protocol. The SMS server runs as a single UNIX process on the SMS database machine. It listens for connections on a well known service port and processes remote procedure call requests on each connection it accepts.

GDB, through the use of BSD UNIX non-blocking I/O, allows the programmer to set up a single-process server that handles multiple simultaneous TCP connections. The SMS server will be able to make progress reading new RPC requests and sending old replies simultaneously, which is important if a reasonably large amount of data is to be sent back. The RPC system from Sun Microsystems was also considered for use in the RPC layer, but was rejected because it cannot handle large return values, such as might be returned by a complex query.

A major concern for efficiency in some DBMS's is the time it takes to begin accessing the database, sometimes requiring starting up a backend process. Since this is a heavyweight operation, the SMS server will do this only once when it starts.

The server performs access control checks on all queries. An access control list (ACL) is associated with each query handle, and with many objects within the database. For instance, to add someone to a list, it is sufficient to either be on the ACL associated with the `add_member_to_list` query, or to be on the ACL of the list in question. In addition, lists, users, machines, and file systems have lists of additional users who are allowed to modify them. The concept of an all powerful database administrator is not necessary with SMS, although could be implemented by having the same ACL for all queries that affect the database.

Because one of the requests that the server supports is a request to check access to a particular query, it is expected that many access checks will have to be performed twice: once to allow the client to find out that it should prompt the user for information, and again when the query is actually executed. It is expected that some form of access caching will eventually be worked into the server for performance reasons.

Input Data Checking

Without proper checks on input values, a user could easily enter data of the wrong type or of a nonsensical value for that type into SMS. For example, consider the case of updating a user's mail address. If, instead of typing "athena-po-1" (a valid post office server), a user accounts administrator typed in "athena-po1" (a nonexistent machine), all the user's mail would be returned to sender as undeliverable.

Input checking is done by both the SMS server and by applications using SMS. Each query supported by the server may have a validation routine supplied which checks that the arguments to the

query are legitimate. Queries that do not have side effects on the database do not need a validation routine.

Some checks are better done in applications programs; for example, the SMS server is not in a good position to tell if a user's new choice for a login shell exists. However, other checks, such as verifying that a user's home directory is a valid file system name, are conducted by the server. An error condition will be returned if the value specified is incorrect. The list of predefined queries defines those fields which require explicit data checking.

Backup

It is not critical that the SMS database be available 24 hours a day; what is important is that the database remain internally consistent and that the data never be lost. With that in mind, the database backup system for SMS has been set up to maximize recoverability if the database is damaged. Backup is done in a simple ASCII format to avoid dependence on the actual DBMS in use.

Two programs (*smsbackup* and *smsrestore*) are generated automatically (using an *awk* script) from the database description file. *smsbackup* copies each relation of the current SMS database into an ASCII file. *Smsbackup* is invoked nightly by a command file that maintains the last three backups on-line. These backups are put on a separate physical disk drive from the drive containing the actual database and copied over the network to other locations. *Smsrestore* does the inverse of *smsbackup*, taking a set of ASCII files and recreating the database. These backups by themselves provide recovery with the loss of no more than roughly a day's transactions. To obtain complete recovery, it is necessary to examine the log files of the servers. An automated procedure to do this has not been written since it is complicated and has not been needed yet.

The Data Control Manager

The Data Control Manager is a program responsible for distributing information to servers. The DCM is invoked by *cron* at regular, frequent intervals. The data that drives the DCM is read from the database, rather than being coded into the DCM or kept in separate configuration files. Each time the DCM is run, it scans the database to determine which servers need updating. Only those that need updating because their configuration has changed and their update interval has been reached will be updated.

The determination that it is time to check a service is based on information about that service in the database. Each service has an update interval, specifying how often providers of that service should be updated, and an enable flag. For each server/host tuple, the database stores the time of the last update attempt, whether or not it was successful, and an override flag. If the previous update attempt was not successful, the override flag will indicate when to try again. The administrative user's interface provides a

mechanism to set the override flag manually, so as to update a server sooner than it otherwise would be updated.

Locking is also provided since an update may still be in progress the next time the DCM is invoked. Without this locking the new DCM process would attempt to update the same service that the older process is still working on.

If it is necessary to update a server, a separate program (also named in the database) is invoked to extract the information from the database and format into the server-specific files. The DCM then contacts the update server on the machine with the target server, sends the necessary data files and the shell script that is invoked on the server machine to install the new files. The success flag is set or cleared based on whether the update attempt succeeded. If the attempt failed, the override flag is set, requesting that another attempt be made to update this server sooner than indicated by the default update interval.

For performance reasons, some parts of the DCM currently touch the database directly rather than going through the SMS server. Nothing is being done that could not be done through the server. However, bypassing the server makes extracting large amounts of information much faster and avoids slowing down the server for these operations.

System Performance

The system is more reliable than the one previously in use at Athena. The old version had an update mechanism more suited to the scale of tens of timesharing hosts rather than thousands of workstations. System crashes have been rare. The speed of the system is fair, being fast enough to use interactively, although some queries do take a while to complete. The database currently occupies about 13 megabytes on the server.

Most of the system as described here has been in use for over six months. A few of the points mentioned here are just now being implemented and put into service. Note that this is the only management tool for 5000 active users, 650 workstations, and 65 servers.

Availability

The SMS server has nearly always been accessible. Some queries, such as listing all publicly accessible mailing lists, will tie up the server for a short period of time. Regular users are prohibited from the longer queries such as listing all users, which will lock up the server for several minutes. The server is occasionally down for safety when an SMS administrator wishes to modify the database directly through Ingres.

Security

Kerberos authentication on all network access and physical security of the machine have been sufficient to prevent breakins. However, the system has not had enough exposure to believe it is really

resistant to concerted attacks.

One problem with the current implementation has to do with security and access control lists. It is difficult to administer the numerous ACL's in the system, and it is not always obvious when different queries are used to predict the effects of changing an ACL. Currently this problem is avoided by having two classes of queries: those that anyone can do, and those that only the database administrators can perform. There need to be more intermediate levels. For instance, it would be useful to allow some system operators to change the information describing public workstations, but not the timesharing machines and service hosts.

Consistency

The current database suffers from decay. The database grows indexes and reference counts that are wrong, post office boxes that do not belong to any user, and groups with no members. These are assumed to be caused by coding errors in the client library and clients. Any problem is potentially compounded by the very *raison d'être* of SMS. With hundreds of workstations in the field there is no guarantee that SMS client programs installed on them are all at current revision level. There are also a few problems suspected in the database system itself (Ingres), but these are difficult to pinpoint. However, the various data extraction programs used by the DCM are robust; they skip over any inconsistent records so these cause few problems.

A database consistency checker, in the spirit of *fsck*, is run regularly, but only for informational purposes. Because of the dangers involved in modifying the database outside the SMS server, it is not modified by this checker at this time.

A shortcoming of the current system is that it is occasionally necessary to use Ingres directly to make some modification to the database. For example, it was recently necessary to modify every instance of a particular login shell in the user account information. A special client could have been written to do this, but it would only be used once, so the operation was done interactively with the Ingres query interpreter. A large number of similar operations to be executed would normally imply a need for a generalized batch processor. This is too complicated for our needs, hence such operations are either done by hand through the regular clients, or edited into a script that can be run through the raw Ingres interpreter. Availability of this interpreter makes such operations relatively easy; yet there is the temptation to fix too many things that way, bypassing the checks built in to the client library and SMS server.

Care must be taken to avoid hardcoding into the SMS design current policy decisions about accounts and resources here at Athena. Yet by maintaining this flexibility, it sometimes is too easy to break the rules. This is more often an error than an intentional breaking of the rules. For instance, there have been

users who did not have a post office box because of administrative mistakes; they were unable to receive mail.

Conclusion

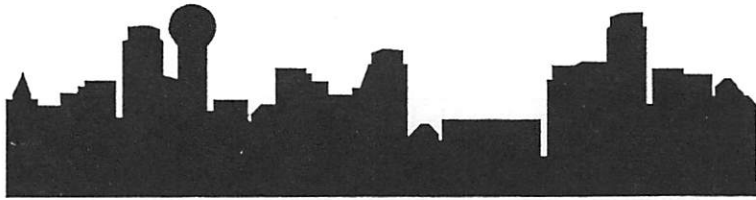
Systems for managing the otherwise unmanageable need to be designed well, burned in realistically, and provided with seamless upgrades. With the Athena SMS, we have an example of a system that is working for the scale of 1,000 workstations, but needs significant refinement to go to the 10,000 workstation level. The initial vision has proven correct; the remaining question is whether the design as we now have it will be capable of the next leap in scale. We believe it will and will be back to report to you with our results.

Acknowledgments

The authors would like to acknowledge the following people from M.I.T. Project Athena for their help in making SMS a reality: Michael R. Gretzinger, a former Systems Programmer, and William E. Sommerfeld, Jean Marie Diaz, Ken Raeburn, José J. Capó, and Mark A. Roman, students working for Athena System Development, who helped with the design and implementation of the system; and Jerome Saltzer, Technical Director of Project Athena, and Jeffery Schiller, Manager of Operations at Project Athena, for invaluable critiques of the design of the system and this paper.

References

1. S. P. Dyer, "The Hesiod Name Server," in *Usenix Conference Proceedings*, (Winter, 1988).
2. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," in *Usenix Conference Proceedings*, (Summer, 1985).
3. J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Usenix Conference Proceedings*, (Winter, 1988).
4. M. T. Rose, *Post Office Protocol (revised)*, University of Delaware (1985). (MH internal)
5. Rand Corp., *The Rand Message Handling System: User's Manual*, U.C.I. Dept. of Information & Computer Science, Irvine, California (November, 1985).
6. N. Mendelsohn, *A Guide to Using GDB*, M.I.T. Project Athena (1987). Version 0.1
7. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Section E.2.1: Kerberos Authentication and Authorization System*, M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).
8. P. J. Levine, M. R. Gretzinger, J. M. Diaz, W. E. Sommerfeld, and K. Raeburn, *Section E.1: Service Management System*, M.I.T. Project Athena, Cambridge, Massachusetts (1987).
9. Relational Technology, Inc., *Ingres Reference Manual*, Release 5.0, Unix 1986.
10. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions On Graphics* 5(2) pp. 79-109 (April, 1987).



The *Zephyr* Notification Service

C. Anthony DellaFera
Digital Equipment Corporation
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
tony@ATHENA.MIT.EDU

Mark W. Eichin
Robert S. French
David C. Jedlinsky
John T. Kohl
William E. Sommerfeld
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
{eichin,rfrench,opus,jtkohl,wesommer}@ATHENA.MIT.EDU

ABSTRACT

Zephyr is a notice transport and delivery system under development at Project Athena.¹ *Zephyr* is for use by network-based services and applications with a need for immediate, reliable and rapid communication with their clients. *Zephyr* meets the high-throughput, high fan-out communications requirements of large-scale workstation environments. It is designed as a suite of "layered services" based on a reliable, authenticated notice protocol. Multiple, redundant *Zephyr* servers provide basic routing, queueing, and dispatching services to clients that communicate via the *Zephyr* Client Library. More advanced communication services are built upon this base.

Introduction

This paper is a brief introduction to the concept of a notification service in general and to the design of the *Zephyr* Notification Service in particular. A notification service provides network-based services and their clients with immediate, reliable, and rapid communication for small quantities of time-sensitive information. The sections which follow address the following issues:

- Motivation for developing a notification service.
- Role of a notification service in networked workstation environments.
- Design constraints.
- Services provided for users by a notification service.
- Unsolved problems and topics for future development.

While this paper is about *Zephyr*, Project Athena's Notification Service, it is our belief that the concepts presented here can be generalized to fit a broad range of notification services and systems. A good understanding of a notification service can be acquired by comparing the *Zephyr* Notification

Service and a more traditional method of workstation message delivery, electronic mail (specifically, *send-mail*). Table 1 makes this comparison.

A Solution To Communication Needs

When services designed for use in a time-sharing environment are used for a very large system of networked workstations, certain communication services begin to fail.[†] They predominantly fail from to their inability to cope with large increases in network scale (i.e., an increase in both the number of workstations and the number of interconnected local area networks). After examining how certain of these services communicate with their clients, we have identified two primary failure modes. These are the inability of a service to cope with increasing server-to-client fanout and the inability of clients to deal by the replacement of a local service with a remote,

[†]Actually, services in general begin to fail, but the scope of this discussion is primarily the realm of communication services. While some of the services, such as *On-Line Consulting*, may be unfamiliar to the reader, they are part of the Athena environment. See the companion paper on Athena Changes to Berkeley Unix, this volume, for an overview of that environment.²

A Comparison Between <i>Zephyr</i> And Mail		
Metric	<i>Zephyr</i>	Electronic Mail
Addressing	Implicit/Dynamic: All addressing is determined dynamically; an explicit "address" is not required. One-to-one addressing is supported by explicitly specifying of recipient ZID. The notice subscription layered service allows for self-selection by the recipient.	Explicit/Static: Sender must know and explicitly provide the name and address (except for "local" mail) of each recipient. Static mailing list support is provided. Only recipients explicitly named receive messages.
Delivery Method	Notices are delivered via dynamically routed reliable datagrams. No connections need be established or maintained. Multiple levels of notice acknowledgment are supported.	Mail is delivered using a point-to-point TCP/IP connection. Acknowledgments are not supported, per se. Return receipts may be requested but are not guaranteed to work.
Delivery Action	Asynchronous/Active: Notices arrive without user intervention. Notices to a particular user are delivered automatically and immediately to all of his or her current login sessions.	Synchronous/Passive: Mail must be sent and read manually by the user. Mail, in general, is delivered to one particular place (a "post office" or "mail drop") for each user; s/he must then actively retrieve it.
Message Length	Short, fixed, notice length with a maximum of approximately 10 lines of 80 characters each.	Long, typically unfixed, message length. Mail messages may be and often are extremely large, on the order of many pages. When message length is fixed it is usually done so by unpredictable rules that vary from site to site.
Message Persistence	Notices are considered time-sensitive; no queuing is provided. If the client recipient is not logged in, then the notice is flushed.	Long time-to-live. Messages typically remain in a mail drop until read.
Message Fan-out	High fan-out: Sending to large lists is efficient. Each client generates one copy of a notice regardless of the number of recipients. Each server receives one copy of a notice being routed regardless of the number of recipients.	Low fan-out. Sending to large lists can consume a great deal of the computing resource. If a message is sent to n users, n copies are generated by the sender, each of which is retained indefinitely by its recipient.
Traffic Performance	High volume/High throughput: Notices may be transmitted in large numbers due to the low overhead of dynamically-routed reliable datagrams.	Medium volume/Low throughput: Large mail messages can send a reasonable volume of data, but slowly (as connections need to be established and routes determined).
System Configurability	Dynamically reconfigurable: Dynamic resource allocation and configuration within the base notification services allows for automatic and simple user-level reconfiguration of layered services.	Statically reconfigurable: Reconfiguration of Unix mail systems is wizard-level work and has significant global impact. No utilities are provided for dynamic system modification or reconfiguration. All changes must be made centrally and atomically.
System Maintenance	Low maintenance: Layered services dynamically recover unused resources through time-outs and reference counting.	High maintenance: Mail requires a post office staff to maintain post office boxes (mail drops), mailing lists, the routing system, and to manually reroute "dead letters."

Table 1

distributed service. The *Zephyr* project was begun to provide a solution to these two failure modes in the context of time-sensitive communications. *Zephyr* has grown into a more powerful tool than was originally anticipated; what began as the development of a "desirable" service soon turned into the development of a "required" service.

The following services are candidate clients of a notification service. All need either the base level *Zephyr* service, which will deliver a message to an identifiable but unlocalized recipient, or the notice subscription layered service, which will deliver a message to the set of potential recipients interested in, i.e. subscribing to, messages of that class. These service levels are discussed more fully in the next section.

File Service A file server can send notices to the users and hosts that it knows would be affected by a change in file server status, e.g. a shutdown. If those users also register a subscription with the notice subscription layered service, other providers of information, such as operations staff, would also be able to reach the user.

Post Office Service Remote post offices can notify users about the arrival of new mail.

Electronic Meeting Service Electronic meeting services (conferencing systems) can notify interested users of new transactions, using the notice subscription layered service.

Print Service Print servers (and queuing services in general) can send job status information back to the job's submitter.

MOTD Service Message-of-the-day information (system wide, service-specific, or local) can be sent to users, such as when they begin using a particular service in the case of a service-specific MOTD.

On-Line Consulting The notification service can be used as the underpinning of a dynamic on-line consulting service. The notice subscription layered service can be used to provide topic based information routing, user location, and consultant-to-client rendezvous.

Host Status Service Broadcast-based host status systems (e.g., *ruptime*) do not scale to a large workstation environment; disk usage grows linearly with network size and total packet computation time grows geometrically. The notification service can provide immediate host status and error log notification on selected hosts or servers.

User Location Service Broadcast-based user location systems (e.g., *rwwho*) also do not scale to a large workstation environment for the same reasons noted above under *Host Status Service*. The notification service can provide asynchronous and immediate user location and state change (login/logout) notification on selected users or groups of users. This can facilitate communication among users. For example, within the

limits of user permission and access control (described in the section *A User's Overview*), students can watch for their friends or development team members for their co-workers.

Talk or Phone Service In a network of workstations, one must know the exact address of others in order to *talk* to them. A notification-based *talk* facility can be constructed that locates the party being called, transmits a talk request notice to that party and, if permission is granted, automatically establishes a *talk* connection.

Emergency Notification In the Athena environment, there is a requirement to provide a simple, asynchronous, and secure means of sending urgent notices to all users on a workstation or in a particular group of workstations. Broadcast methods are not useful on a large scale and are by definition imprecise. In addition, the workstation user must trust the broadcasting host and the person issuing the message. Using *Zephyr*, emergency notices can be sent directly to all users on any specified host, with authenticity[†] guaranteed.

Message Service Current *write* services suffer the same problems noted above under *Talk or Phone Services*. A notification-based *write* utility is trivial, since almost all the work is subsumed by the notification service. Write notices can go to individual users or to previously created subscription groups.

Other Service Events The notification service can be used to reliably notify users of a wide range of asynchronous service events that occur in distributed workstation environments. This notification can be accomplished by using the base notification service, the emergency notification service, or one of the notification service layered services.

Fitting The Tool To The Job

Zephyr is designed around a system of dynamically-updated, locally-authoritative servers that provide centralized routing, queuing, and dispatching. Clients communicate with these servers via the *Zephyr* Client Library interface. The *Zephyr* Client Library implements the *Zephyr* Protocol, a reliable, authenticated notice transmission protocol. In our design we view the notification service as a suite of "layered services" built upon a base notice transport layer. Additional layers provide more advanced communications services. This design is analogous to that of the X Window System.⁴

Zephyr notices consist of two parts: a routing header and client data. It is the job of the *Zephyr* Servers to route notices between *Zephyr* clients based upon attributes specified in the notice's routing

[†]Since *Zephyr* relies on authentication, it is also suggested that you read the *Kerberos* paper in this volume.³ This provides a general introduction to the Project Athena *Kerberos* Authentication System.

header. Servers do not examine a notice's client data; it is entirely possible that that data is encrypted or otherwise uninterpretable. By examining the attributes in a notice's routing header, a *Zephyr* Server computes the list of recipients of a notice. The most basic routing attribute that may be specified is a single recipient, named by a ZID.[†] More complex routing attributes are specified for the layered services. The notice classification information for the notice subscription layered service discussed above or specialized keywords for use with a rule-based routing service are examples of such complex attributes.

Determining notice recipients based upon routing header attributes is known as "subscription multicasting". Subscription multicasting is a passive routing technique; attributes not recognized by a service layer are simply ignored. This allows layered services to implement different notice routing methods that peacefully co-exist while using the same base notification service. In this way subscription multicasting differs from other message routing techniques such as network broadcast or *sendmail*. Because the set of recipients for any notice can always be determined, it is more efficient and less vulnerable to increases in network scale than broadcast techniques. Because additional resources, routing methods, and recipients may be dynamically added by almost any user, it requires less maintenance and incurs less overhead than traditional list based message transmission techniques (such as *sendmail*).

Zephyr clients determine what level of service they receive from the notification service by choosing the service layer with which they communicate. For example, certain system services have complete knowledge of their clients, and only need the notification service to route information to those clients. This is the most basic service layer. For example, a file server knows the particular users it is serving, and needs only the ability to reliably notify those users about service state changes. On the other hand, client services that cannot identify their clients (or may simply not know who is interested in such state information) may wish to notify "interested parties" about service state changes. A workstation error logger would fall into this category. This type of service would make use of the notice subscription service layer. Such a service layer provides the ability to store communication state information for client services external to those services. This adds to the notification service the unique ability to provide to its clients status and availability information about other services even when those services are disabled and cannot communicate with their own clients.

[†]Recipients must be uniquely identifiable. *Zephyr* relies on *Kerberos* to both provide and guarantee these identifiers. So as to avoid confusion with the sense of a UID, we shall refer to the *Zephyr* identifier as a ZID.

Design Requirements And Constraints

The goal of the *Zephyr* development is to produce a 4.3BSD Unix implementation of a notification service useful to Athena. This implementation should consider the effects of:

Scale - Such a service must efficiently provide its capabilities with the highest possible fan-out (i.e., client to server ratio), without adversely affecting network load or server host performance. Additional redundant servers must be easy to install, and must provide load sharing immediately.

Evolution - Since *Zephyr* is an evolving service, it is must gracefully handle protocol compatibility from one version of the service to the next.

Management - It must be possible to perform all aspects of service maintenance and operation remotely.

Network Protocols - Since the notification service depends upon an underlying network transport mechanism, it accepts those design constraints imposed by that mechanism. The *Zephyr* Protocol, as currently implemented, is based on the Unreliable Datagram Protocol (UDP). As such, it is constrained to operate within the capabilities of UDP. However, there is nothing in the design of the protocol that would prevent its using other network transport mechanisms (such as a remote procedure call system). Constraints imposed by UDP are listed below, along with a brief description of how they affect *Zephyr* application programmers and end users.

Duplicate Notices UDP does not provide any suppression of duplicate packets, *Zephyr* clients may receive duplicate *Zephyr* notices. *Zephyr* applications must be capable of dealing with this possibility.

Missequenced Notices UDP does not provide packet sequencing. While *Zephyr* notices do contain timestamps, it is up to the application to check the timestamp and handle notices received out of sequence.

Flow Control UDP does not provide any flow-control capability. *Zephyr* applications must be capable of dealing with notices at whatever rate they arrive or be willing to lose notices.

Unreliable Delivery UDP does not provide a reliable delivery mechanism. *Zephyr* does provide several levels of acknowledgment processing, but it is up to the application to decide how much overhead it is willing to incur in order to guarantee notice delivery.

Packet Size UDP packets have a relatively small, maximum size. Considering the amount of routing data and other information that *Zephyr* must store in each packet, this becomes a constraint on how much user data may be included with each packet.

How visible these constraints are to the end user is up to the *Zephyr* application programmer. For

example, our *zwrite* application (which allows users to exchange *write*-like messages) only guarantees that the message was sent, not that it will actually arrive or how many copies will arrive.

In order to provide ZID-based communications, the notification service must dynamically maintain a database that maps ZID's to their current physical locations in the network. The only requirement on the ZID used in addressing is that it must be a network wide ZID and be "registered" as a client of the notification service (i.e., have its physical address(es) stored in the notification service database). In particular, *Zephyr* manages a location database that maps *Kerberos* "principal names" (authenticated user names) into a tuple of physical location information (geographic location, hostname, IP port number, and tty, among other things). This database is primarily used by *Zephyr* for notice routing, but is also made available to *Zephyr* clients via the User Location Layered Service discussed below.

The reliability of the information stored in the user location database imposes constraints upon applications that rely on *Zephyr*.

- User location information present in the database can be assumed to indicate that a user has logged in, because user logins that are reported to *Zephyr* must be *Kerberos* authenticated.
- User location information present in the database cannot be assumed to indicate that a user is still logged in, because there is no way to guarantee an orderly user logout. (For example, a workstation may crash).
- The absence of user location information in the database cannot be assumed to indicate that a user has not logged in, because a user can choose to not make his or her login information publicly available.

Zephyr attempts to prevent user location data from persisting when it is no longer valid. If a workstation crashes, the user login sessions on that workstation are necessarily terminated without sending logout notices to a *Zephyr* server. This implies that logins in the database may not always be valid. To cope with this, a specialized *Zephyr* client runs during the workstation reboot sequence. This client simply tells a *Zephyr* server to flush any previous state information associated with the (rebooted) workstation.

A User's Overview

For this discussion, a "user" of *Zephyr* is either a user of a *Zephyr* client or an applications programmer who is using the *Zephyr* Client Library.

The *Zephyr* system can be viewed as divided into two parts, clients and servers. There must be at least one *Zephyr* Server (*zephyrd*) per *Kerberos* realm (realm of authority of a particular authentication service), one *Zephyr* HostManager Client (*zhm*) per active workstation and one *Zephyr* WindowGram Client (*zwgc*) per user login session. To ensure reliable

service, there should be several *Zephyr* Servers per *Kerberos* realm.

When a workstation is reboots, a *zhm* is automatically started. The *zhm* serves two purposes. First, it acts as a reliable transmission tower for notices sent from local *Zephyr* clients. Second, the *zhm* acts as an emergency notice routing channel on the individual workstation. When *zhm* starts up it first seeks out a *Zephyr* Server and registers itself with that server. From then on, that *zhm* and all clients that communicate through it are "owned" by that server. Only that server will be considered to have "authoritative" information about *Zephyr* clients on the workstation managed by that *zhm*.

All *Zephyr* clients on a workstation use the *Zephyr* Client Library to send and receive *Zephyr* notices. The *Zephyr* Client Library routes all notice traffic leaving a workstation through that workstation's *zhm*. In this way, clients themselves are not required to have to locate and manage communications with a *Zephyr* Server. If the *zhm* loses contact with its *Zephyr* Server (i.e., the *Zephyr* server which owns it does not respond within a fixed but configurable safety margin) it is the *zhm*'s job to seek out and contact a new *Zephyr* Server.

When a user logs into a workstation, a *zwgc* for that user is automatically started, provided that the user can provide an authenticator and that the user has not deliberately disabled *Zephyr*. If the user is not interested in using *Zephyr*, it is still important that s/he have a *zwgc* running. The most important reason is that *zwgc* is the contact point for *Zephyr* emergency notices. These notices are transmitted by certain privileged users (e.g. operations staff), *directly* to the workstation's *zhm*. The *zhm* then forwards these notices to all *zwgc*'s currently running on the workstation.

When the *zwgc* starts up, it registers the user with *Zephyr* and, depending upon the setting of certain *Zephyr* control variables, may make other *Zephyr* requests. These variables may be modified using the *Zephyr* Control Utility (*zcctl*). The most important of these variables is the *Zephyr* exposure level variable. This variable controls how much information about an individual user *Zephyr* will store and make available to requesting clients. There are currently six possible settings for this variable:

none – This completely disables *Zephyr*. The user is not registered with *Zephyr*. No user location information is retained by *Zephyr*. No login or logout announcements will be sent. No system default notice subscriptions will be entered for the user.

opstaff – The user is registered with *Zephyr*. Only system operation notices and emergency notices will be received. No user location information is retained by *Zephyr*. No login or logout announcements will be sent. System default notice subscriptions will be entered for the user.

realm-visible – This is the default exposure setting.

The user is registered with *Zephyr*. All notices will be received. User location information is retained by *Zephyr* and made available only to users within the *Kerberos* realm. No login or logout announcements will be sent. System default notice subscriptions will be entered for the user.

realm-announced – The same as **realm-visible**, plus login/logout announcements will be sent to users within the *Kerberos* realm who have explicitly requested them).

net-visible – The same as **realm-visible**, plus user location information is made available to any user who requests it.

net-announced – The same as **realm-announced**, plus login/logout announcements will be sent to any user who has requested them.

zwgc is a vital client. For this reason *zwgc* has two primary interfaces. The first, and most powerful, is an X Window System interface referred to as a “WindowGram browser.” This browser allows the user to scroll through and perform certain operations (such as “save”, “delete”, “cut” and “paste”) on all the notices that *zwgc* has received. If a user logs into an X display, *zwgc* selects this interface. If the user does not have access to an X Display, *zwgc* selects a simple terminal based interface.

In addition to the basic services described above, *Zephyr* provides additional layered services that are built on the base notification service.[†] The two most important of these are the *Zephyr* Notice Subscription Layered Service and the *Zephyr* User Location Layered Service.

The *Zephyr* Notice Subscription Layered Service provides a dynamic information dispersal service based upon a “subscription list” paradigm. The most important use for this layered service is by services that cannot directly identify their clients or may simply not know who is interested in the information they are providing. Such services may wish to simply send notices to “all interested parties.”

The simplest function that the *Zephyr* Notice Subscription Layered Service provides is a method for users and groups of users to exchange notices. This is accomplished with the *zwrite* utility. In addition to person-to-person messages *zwrite* allows users to send notices to notice subscription lists.

Restricted access to user location information is made available to *Zephyr* clients by the *Zephyr* Client Library. This information is dispersed by the User Location Layered Service. The database which this layered service uses is maintained internally by *Zephyr* to track the existence of ZID's in the network. A user can be located through *Zephyr* by using the *zlocate* and/or *znol* utilities. These utilities make calls to

the *Zephyr* User Location Layered Service for the user. The *zlocate* utility allows users to manually locate one or more users. The *znol* utility makes use of the *Zephyr* Notice Subscription Layered Service to subscribe to user login/logout notices from a list of ZID's provided by the user.

When a workstation crashes, all clients running on that workstation are lost. When this happens, client state information that *Zephyr* has associated with that workstation must be flushed and any corresponding *Zephyr* system resources must be freed. This is done in two ways. The first occurs slowly while the workstation remains down, the second when it reboots. If the workstation remain down long enough all invalid state information will be incrementally detected and flushed. This incremental flushing occurs whenever a *Zephyr* Server attempts to send (or route) a notice to a client and discovers that the workstation on which that client is supposed to be running is not responding. When the workstation does eventually reboot, *zhm* is called with a special “reboot flush” flag. This causes *zhm* to run just long enough to transmit a special workstation state flush notice to the first available *Zephyr* Server. These two “garbage collection” techniques work together to keep the *Zephyr* system's database current.

The last phase of user interaction with *Zephyr* occurs at logout time. When the user logs out, *zwgc* notifies *Zephyr* to flush all state associated with that user's login and then exits.

The *Zephyr* system currently consists of the following suite of programs:

<i>zctl</i> (1)	<i>Zephyr</i> subscription control program
<i>zinit</i> (1)	<i>Zephyr</i> login initialization program
<i>zleave</i> (1)	Remind you when you have to leave
<i>zlocate</i> (1)	Find a user
<i>znol</i> (1)	Notify on login/logout of specified users
<i>zwgc</i> (1)	<i>Zephyr</i> WindowGram Client
<i>zwrite</i> (1)	Write to another user
<i>zhm</i> (8)	<i>Zephyr</i> HostManager Client
<i>zephyrd</i> (8)	<i>Zephyr</i> Server daemon
<i>zstat</i> (8)	Display <i>Zephyr</i> statistics

Future Directions And Unsolved Problems

Once the basic notification service is in place it becomes a simple matter to provide many other layered services based upon it. The *talk* service mentioned above is a good example of a service that utilizes multiple *Zephyr* service layers. We envision *Zephyr* as a transport service that can incorporate new notice routing methods as they are developed. *Zephyr* is designed to allow communication development efforts to occur side-by-side with running production systems that utilize *Zephyr*. For example, researchers at M.I.T.'s Sloan School of Management have expressed interest in using *Zephyr* as the transport service layer for a rule based communication system.

The following is a list of some of the areas of future development. They are either unsolved

[†]The architectural design details of the *Zephyr* Service Layers are discussed in the *Zephyr* design document.⁵

problems or areas that need further investigation.

- Extend the *Zephyr* Protocol for use across long-haul networks.
- Modify the *Zephyr* Server to allow ZID registration from other *Kerberos* authentication realms.
- Modify the *Zephyr* Server to forward *Zephyr* notices to recipients in other *Kerberos* realms.
- Develop a more formal interface definition for use between the *Zephyr* notice transport layer and *Zephyr* Layered Services.
- Develop a more advanced user interface for the *Zephyr* WindowGram Client.
- Decouple *Zephyr* from the *Kerberos* system. In the current implementation, they are linked together too closely.
- Integrate with more clients.

Conclusions

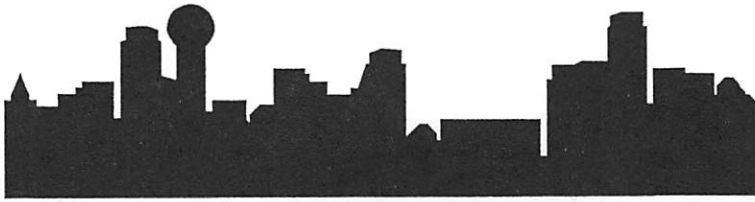
Zephyr has proven useful in providing a mechanism for transporting time-sensitive information in a large-scale workstation environment. It not only permits existing services from the timesharing world to evolve toward the workstation world, but also permits new services to grow alongside. It makes reasonable compromises between reliability and complexity and is already of use to both users and operations staff. Indeed a major problem has been its popularity while still under development.

Acknowledgements

The authors would like to acknowledge the following people from M.I.T. Project Athena for their help in making *Zephyr* a reality. Michael R. Gretzinger, former Systems Programmer, and David G. Grubbs, former Manager of Systems Integration, for their suggestions on the initial concept of a Notification Service, and Daniel E. Geer, the Manager of Systems Development, for his undying support of our efforts. We also thank Katharyn L. Lieben and G. Winfield Treese for the improvements they made to this paper.

References

1. E. Balkovich, S. R. Lerman, and R. P. Parmelee, "Computing in Higher Education: The Athena Experience," *Communications of the ACM* 28(11) pp. 1214-1224 ACM, (November, 1985).
2. G. W. Treese, "Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD," in *Usenix Conference Proceedings*, (Winter, 1988).
3. J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Usenix Conference Proceedings*, (Winter, 1988).
4. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions On Graphics* 5(2) pp. 79-109 (April, 1987).
5. C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, *Section E.4.1: Zephyr Notification Service*, M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).



Ralph R. Swick
Digital Equipment Corporation
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
swick@ATHENA.MIT.EDU

Mark S. Ackerman
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
ackerman@ATHENA.MIT.EDU

The X Toolkit: More Bricks for Building User-Interfaces or Widgets For Hire

ABSTRACT

Primitives for application-level user interface construction facilities currently under development at M.I.T. Project Athena are described. The design philosophy of the X Toolkit and associated widgets and some of the practical implications are discussed.

Introduction

The X Window System^{†1,2} was developed at the Massachusetts Institute of Technology to satisfy the needs of a broad spectrum of users for a high-performance, high functionality, network-based window system that can be implemented on a wide variety of high-resolution raster graphics display devices. The widespread interest and unprecedented vendor support for the X Window System has assuaged one of the principal concerns of application developers: the cost of supporting multiple hardware platforms with different base technologies, including window systems.

The X Window System has been carefully designed to address two (sometimes conflicting) desires of application developers: to use hardware-level techniques for maximum performance and to maintain portability with a common programming interface across multiple vendor platforms. X has succeeded in gaining broad vendor support largely because its specification is intentionally restricted to the set of primitives needed to manipulate multiple independent window contexts on raster graphics displays without declaring (or restricting) the choice of particular user-interface semantics. It is explicitly intended that developers be able to choose a visual interface appropriate to their needs, their corporate philosophy, their research requirements, religious preferences, or whatever.

[†] The X Window System and X Windows are trademarks of the Massachusetts Institute of Technology. Use of the latter is strongly discouraged. The developers prefer simply "X" when a shorter form is required.

This restriction to low-level control and input primitives in the definition of the X communications protocol and in the corresponding application interface layer, Xlib³, is either a strength or a weakness in the X Standard, depending on the reviewer's point of view. Vendors whose installed base of products contains well-defined visual interface and human-computer interface researchers find this flexibility in the standard to be of major importance. However, many application developers consider user interaction and other higher-level graphics libraries to be a base system technology that should be provided by the hardware vendors and, of course, be standard across vendors.

To address the desires of such developers for common higher-level development tools, there are several projects under way at various locations covering different application needs and problem domains. One such project is the X Toolkit project, a collaborative effort of MIT/Project Athena, DEC/Western Software Laboratory, Hewlett-Packard Company/Corvallis Workstation Operation and others. The X Toolkit project is producing an applications interface layer above the Xlib layer specifically tailored to visual user interface construction.

Toolkit Overview

The X Toolkit (hereafter called simply "Xtk") recognizes that no single comprehensive set of user interface tools is likely to be acceptable for standardization in the near future. In order to maximize the utility and acceptability of the user interface library, Xtk has been divided into two separable pieces. These

two layers will be described below.

The fundamental entity in Xtk for user interface construction is the *widget*.†

The core of Xtk, the “Intrinsics” (a term appropriated from a previous H-P user interface library for X), is a set of utility routines intended for use in developing widgets. The Intrinsics are a set of user interface primitives that are themselves free of visual and interaction style. The Intrinsics do not constrain the widget writer to make the widget look or operate in any particular way. These primitives may be used together or separately to produce higher layers which do incorporate specific policy and style. Such higher layers will further reduce applications development cost.

Most applications will call only a few of the Intrinsic routines directly. These routines offer a uniform programming interface to the basic procedures (*methods*) of all widgets, regardless of the widget type.

The Xtk Intrinsics have been presented in an earlier paper⁴ and, although the detailed design has evolved,⁵ the philosophy and architecture of the X Toolkit remain the same. The principal additions are a class hierarchy for widget types; the separation of widget identifiers from the corresponding X window identifiers; and the ability, using the class hierarchy, for new widgets to inherit methods from an existing widget. These changes simplify significantly the task of widget development and make widgets more modular.

Widgets define input semantics and visual appearance. Some widgets are pliable; their input semantics (mouse buttons, pointer motion, keyboard input) are bound at run-time, while other widgets may have fixed (hard-coded) semantics. Likewise, visual appearance (highlighting, repositioning, or other animation) may be fixed or may be adjusted at run-time.

The Intrinsics provide a uniform way for widget developers to handle the common chores of widget construction: initialization, input event dispatching (including enabling and disabling user input dispatch to sub-hierarchies of widgets), run-time configurability, uniform handling of common events (such as exposure and re-size), cleanup, and others. The Intrinsics also include the uniform application programming interfaces for creating, controlling, and destroying widgets. The Intrinsics currently consist of over 90 public procedures, of which half are intended solely for widget construction.

The goal of the Intrinsics is to make possible the quick development of widgets. Sets of widgets should adhere to a consistent application interface, user interaction policy, and visual appearance. It is viewed

as desirable (or, by some, a necessary evil) that the construction of multiple such sets (“widget families”) covering different philosophies be possible with the Xtk Intrinsics.

The second piece of the Xtk is a set of basic widgets. Most application developers require a minimum set of widgets as a component of any product-quality user interface library. The X Toolkit project also recognizes the need for concrete examples in a real widget family. To serve both these needs, the first author is leading the effort at Project Athena to produce a basic widget set that will be included with the version 1 release of the X Toolkit. To distinguish this set from others which we know of, or expect to be developed, we shall here call these the “Athena Widgets”.

In addition to the goal of being a basic widget set, the Athena Widgets have another goal arising from code which had been written prior to X Version 11. Many of the components of Xtk had been prototyped in a toolkit for X Version 10 that was released by DEC Ultrix Engineering in the spring of 1987. The Athena Widgets borrow heavily from those prototypes in order to ease some of the porting burden for certain applications built on these prototypes.

The widgets described here are being developed in conjunction with a set of visual courseware projects at Project Athena. These projects vary considerably in their user dialogs and yet require a standard visual appearance. This has led to an emphasis in the Athena Widgets on handling text, graphics, and video in a variety of ways, and has extended the widget hierarchy to fulfill these needs.

The Athena Widgets are intended to fulfill 80% of application requirements. We have tried to select the critical widgets that will allow the easy solution of individual requirements. (See the section on Creating New Widgets for more on this.)

Intrinsics

One of the principles espoused in the design of the Xtk is the construction of widgets from primitives. We will describe two independent facilities available in the Intrinsics for such construction: subclassing and composition. From an application point of view, every widget is a single object. The actual semantics and appearance of the widget may, however, be very complex. For example, a “control panel” widget is likely to consist of simpler widgets with a “geometry manager” controlling the spatial relationship between the component widgets and possibly a “focus manager” controlling the dispatching of user input to those components. Depending upon the needs of the application, such a compound (or “composite”) widget may be implemented independently and added to the widget library as a new widget class, may be constructed by the application at run-time with in-line calls to the Xtk Intrinsics, or may be constructed by the composite widget itself from a

†We chose this term since all other common terms were overloaded with inappropriate connotations. We offer the observation to the skeptical, however, that the principal realization of a *widget* is its associated X *window* and the common initial letter is not un-useful.

resource record retrieved through the Intrinsic resource management facilities.

Composition of widgets is most appropriate when there are distinct visual regions to a widget, each having separate input/display semantics, and especially when the same semantics may appear in a region of another widget class. In this case, the semantics common to both regions may be extracted into a more primitive widget class.

This is the principle of modularity of widgets: the application will still view a composite widget (a control panel, for example) as a single widget. The internals of this composite widget are built when the widget is instantiated. The composite widget may determine and instantiate all of its components, as for a custom application panel. The components may also be instantiated and assigned by the client of the composite widget. Some of the Athena Widgets exhibit this recursive construction behavior; e.g. Dialog, while others are 'boxes', or frames into which the client inserts independently instantiated widgets, e.g. Form and VPaned.

The second construction facility, subclassing, allows a widget class to semi-automatically inherit some or all of the characteristics of an existing widget class, and to share portions of the code that implement the parent (super-) class methods. The new widget may call upon the superclass methods to manipulate any part of the widget state defined by the superclass and needs only to implement the code to manage the state that is unique to the new class.

The subclassing facility is most appropriate for new widgets which need only to add additional semantics to an existing widget, or to constrain in some manner the full generality of an existing widget. Examples of both subclassing and composition in the Athena Widgets are described below.

Several widget classes have been defined solely for the purpose of being subclassed. The **Composite** class provides methods to maintain a list of child widgets, to manage the insertion and removal of children, to manage requests from the children for new geometries, and to manage the assignment of input events to specific children (input focus). The **Constraint** class has all the methods of Composite and in addition provides methods to automatically create and initialize an arbitrary data record attached to each child. The contents of this data record are defined by each subclass of Constraint and are intended to contain layout information used by the subclass geometry manager. Neither Composite nor Constraint are intended to be instantiated; only their subclasses are. Nothing in the implementation, however, will prevent an application from instantiating any class, should it prove useful.

All widgets are expected to be self-contained with respect to exposure, resize and input event handling. That is, the clients of the widget (i.e. the application program or a composite widget of which this

widget is a component) are guaranteed that all exposure and input events sent by the X server for the window defining the widget will be processed completely by the widget. A client that creates an instance of the ScrolledAsciiText widget, for example, is not involved in any of the details of text re-painting, scrolling, selection, cut and paste, and so on. The client is also free to assign any shape to the widget and assume that the widget will adjust to the imposed size.

The principal mechanism the Athena Widgets use to communicate back to the client is the callback procedure. While a client has the option to query the widget state, it is usually more convenient for a command button, for example, to directly call a client-supplied procedure when 'pressed' by the user. Some widgets accept more than one callback procedure for alternate interactions that they implement.

Runtime Configurability

One of the major design principles followed by the Athena Widgets is to make as much of the user interface as possible customizable by the end-user of the application. Fierce debates (i.e. *wars*) break out every time someone proposes a single set of key or button bindings for all users, or that a fixed choice of colors or text fonts will be appropriate for all individuals. Even such characteristics as whether scrollbars go on the left or right (or top/bottom) of a window may be appropriate for individual customization.

The Xtk implements run-time configurability through the Xlib Resource Manager. The Resource Manager is a general-purpose repository for storage and retrieval of arbitrary data within a single process address space. During initialization, the Xtk preloads the resource database from the X server and from one or more files. When a new instance of a widget is created by the application, the widget resource list is examined and the widget instance is initialized with data from the resource database. Each widget declares an instance name and a class name for purposes of matching against resource names in the database. The Resource Manager defines rules for partial name matches so that a single resource entry may initialize many widget instances.

The implementor of a new widget has the choice of which widget characteristics to declare as resources and which to hard-wire into the widget. In general, any instance data for which the widget is willing to allow modification requests from the client should be declared in the resource list.

Widget characteristics such as text font and color are obvious resource choices. The Athena Widgets also declare the keyboard and mouse button bindings as resources. In this way, the user of any application linked against the widgets has the option to accept our default bindings or enter his/her own bindings. The Resource Manager naming mechanism allows the user to attach the new bindings to all instances of a widget class (e.g. Scrollbar) in any application, or to a specific widget in a specific application only, or any

combination in-between. A client may, if it so chooses, override any entry in the resource database either by storing its own entry (preferred), or by passing an explicit value when instantiating the widget (discouraged).

The keyboard and button bindings are configurable in yet a second way. Each widget that accepts user input declares a list of action routines that may be invoked by input events. The Athena Widgets use the Translation Management facilities in the Intrinsics to bind keys and buttons to widget action routines. These bindings can specify parameters to the action routines to further configure their behavior. The Scrollbar widget, for example, declares three action routines, one of which is parameterized so that the range of values it returns may be established by the bindings.

Using these action routines and the default scrollbar bindings, the ScrolledAsciiText widget, for example, allows the user to scroll a block of text forward or backward by a variable amount and to drag the thumb (elevator) to a new position, displaying any portion of the text. A user may supply an alternate set of Scrollbar bindings that will cause the scrollbar to report full-length forward or backward scrolls, independent of the pointer position, possibly disabling the variable scrolling as desired.

Two of the Athena widget classes exist for the purpose of handling interprocess interactions with other X client processes. The **Shell** widget defines no user action routines, but maintains all the appropriate window properties established by convention for X window managers, including icon representation. Many of these parameters are controlled by command line options and parsed by the Xtk initialization routine. Most applications use a Shell widget as their outermost (top level) widget.

Additional semantics appropriate for temporary, or "pop-up", panels are added to a subclass of Shell, the **Popup** widget. From the client's point-of-view, both Shell and Popup are simple widgets; they manage exactly one child widget and have a trivial geometry manager. UIMS developers may find it desirable to extend Shell at some time in the future, even allowing site tailoring for specific choices of window manager. One such addition might be a Shell that, when made smaller by the window manager (as instructed by the user, of course); added scrollbars (or other interaction semantics); and provided a movable viewport on the application window which, from the application's point-of-view, retains its original size.

Current Widgets

The Athena Widgets are divided into two major classes. The first are simple widgets: various sorts of buttons, labels, edit buffers and the ubiquitous scrollbar. These form the elemental building blocks of a user interface. The second are composite widgets: scrolled text, dialog boxes, and various methods of

putting together simple widgets in more complex arrangements.

All simple widgets have initialization, realization, display, and interaction methods. These methods may be fairly simple, as with the button widgets, or quite complex, as with the text widget.

All widgets use the Core widget as the root of their class hierarchy. The Core widget (whose class name, as a special case, is just **Widget**) has the minimal set of instance fields common to all widgets: width, height, border width, and so on. While it is not usually intended that an application ever create an instance of (Core) Widget, it is supported: an application that wants a simple window within a widget panel *may* instantiate Widget and use the resulting window.

The **Label** widget is just that; a widget that displays either a text string or (in the near future) a pixmap without any interaction semantics and therefore without any callback procedures. It can only center, right justify, or left justify its text in the client's choice of fonts within its assigned size. (The default size is a bounding box for the text or graphics.) A Label may be insensitive, or grayed-out, and the border, as for all widgets, need not be visible. Like all widgets, the Label widget processes exposure events on its window so the parent can be assured that all visible portions of the Label are properly refreshed on the screen. One will typically use the label to position and paint items intended for display only.

The **Command** button widget is a subclass of Label with interaction semantics and therefore a callback procedure. A Command button is a Label with *enter*, *leave*, *set*, *unset*, and *notify* actions. When

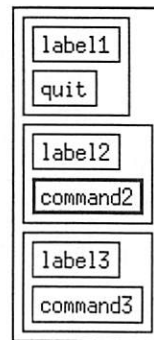


Figure 1

insensitive, the Command button does not respond to user input events. On *enter* to a sensitive Command, the border will, by default, become visibly thicker. On *set*, the button is displayed in reverse video; on *unset*, the button is returned to normal. On *notify*, a single client-supplied callback procedure is activated. These actions are mapped, by default but not by necessity, to the pointer enter, button down, and button up events. One will typically use the Command button to create "clickable" icons or labels that start

a program action. The Command button is an essential building block for point-and-click interfaces.

Figure 1 shows several Labels and Command buttons. The user has moved the pointer into one of the Command buttons, and the button has been highlighted. These Labels and Command buttons are enclosed in two layers of Boxes, which will be discussed below.

The Command widget can be extended into its subclass **TwoState** button. On *set*, the TwoState widget displays a second label or graphic. In all other respects, it is similar to Command. TwoState is useful for simultaneously displaying and changing a binary application state.

The **Toggle** widget is also a subclass of Command and also has simple interaction semantics and a callback procedure. A Toggle is a binary state button. Once *set*, e.g. by button down, it remains in that state until the user selects the Toggle again. Toggle is useful for selecting on-off options. The application may use the callback to be notified on each state change, or may query the Toggle for its current state as appropriate.

The **Grip** widget is a minimal Command button. Grip (a.k.a. Knob) has a single callback but no default interaction semantics. The interactions (event bindings) are provided by the client when Grip is instantiated. Grip simply uses the X window background as its display. This widget was built for clients who need to identify specific locations where, for example, the pointer may be used to drag an object.

The **Scrollbar** widget implements a vertical or horizontal scrollbar. There are three callbacks: one to move the scrollbar thumb (elevator), a second to execute a client callback passing a distance in pixels as data, and a third to execute an application callback passing as data the position of the pointer as a percentage of the scrollbar length. The inclusion of the *moveThumb* action routine in the actions table allows the client some control over whether or not the scrollbar widget automatically repositions the thumb on input.

The **AsciiFile** and **AsciiString** text widgets allow the entry, modification, and display of text. The widgets can be instantiated in read-only or read-write modes. They implement a subset of the ever popular Emacs, including mouse-driven cutting and pasting. Key bindings are, of course, completely settable through the Translation Manager.[†]

The **AsciiString** widget operates on a single in-core text string; **AsciiFile** on a disk file. They are both subclasses of **Text**, which implements most of the

user interactions. The implementation of the **Text** widget follows a source/sink model, allowing for the development of additional text sources (other than string and file) and for additional display sinks.

These few simple widgets are an attempt to create a number of general widgets that can then be tailored for individual uses. A more complex button widget (for example, one that might display two lines of text) would require a different display method but keep the interaction method for Command.

However, few interfaces of any real complexity or use could be created using just these simple widgets. The simple widgets must be combined using Composite widgets. Composite widgets require geometry, child insertion and deletion, and input focus methods. The geometry manager for the composite may or may not be constraint based, and must handle resizing events by re-positioning the child widgets. Generally, the child insertion, child deletion, and input focus methods will be inherited from the superclass, Composite.

The **Box** widget arranges its children in the minimum bounding box. The children are automatically rearranged when one is deleted or added. One can create button areas or horizontal menus using Box.

The **RadioButtonBox** widget is a subclass of Box exclusively for Toggle Buttons. It allows only one Toggle button to be set at a time. If a second Toggle becomes set, the RadioButtonBox will check its list of children and unset any others. This is useful for mutually exclusive application options.

The **Form** widget can contain any number of simple or composite widgets. Form is a general-purpose constraint-based layout widget that can be instructed to maintain fixed separations between children, or to maintain fixed distances between children and the edges of the form, and so on. When Form is resized, it uses the constraint information to resize and reposition the children to maintain the assigned separations. Forms are useful for creating arbitrarily complex interaction windows that exhibit 'nice' resizing behavior.

The **Dialog** widget is a subclass of Form. It arranges a specific combination of Label, AsciiString field, and Command buttons. The Label is displayed on the first line, followed by the text field, and the buttons are on the last line. Dialog is principally a convenience widget implemented to handle a common programming problem.

The **ScrolledAsciiText** widget contains a scrollbar and an AsciiString or AsciiFile widget. Using the default scrollbar bindings, the ScrolledAsciiText widget allows the user to scroll the contained text forward or backward by a variable amount. By dragging the thumb to a new position, s/he can display any portion of the text.

The **Paned** widget manages any number of simple or composite widgets in a tiled manner. The

[†]It has been suggested that some users would prefer an AsciiFile widget that allowed them to specify their favorite text editor. While the full semantics of the Text widget may be difficult to support with an arbitrary external process, it may be an interesting exercise to implement a sub-process interface widget that, in carefully defined circumstances, could be substituted by the user for the AsciiFile widget.

current Paned widget, **VPaned**, stacks its children vertically with the top and bottom edges of successive widgets touching. VPaned uses Grip to allow the user to re-position the boundaries between the tiled widgets.

Figure 2 illustrates the current Athena Widget hierarchy. This diagram should only be of interest to widget developers; application developers should be concerned only with the external characteristics of a widget. The fact that a Toggle button actually uses the Label code to display its text string should be of no consequence to the application. The widget class **Simple** contains only the procedure to change a widget's borders to indicate sensitivity or insensitivity.

Special-purpose Widgets

The **Clock** widget displays an analog or digital time-of-day clock. The **Load** widget displays a continuous system load graph. Both of these were exer-

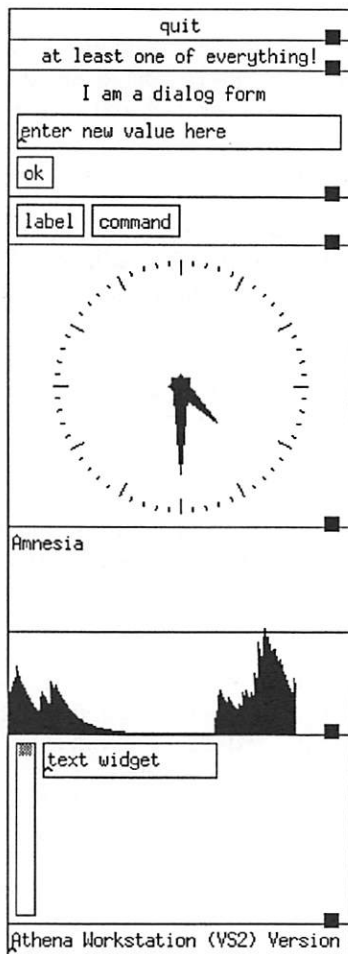


Figure 3

cises in converting existing simple applications into widgets. Load allows the client to supply its own procedure to fetch data, or to use the built-in default *GetSystemLoadAverage* procedure.

Example

Figure 3 shows a variety of composite and simple widgets instantiated in a single trivial application. The outermost widget is a VPaned (inside a Shell) which, in turn, contains several other composite and simple widgets. The small solid boxes are Grips which allow the user to move the attached pane boundary up or down, forcing the associated widgets to be resized.

The uppermost pane of the VPaned is a Command button marked "quit". Appropriately, when the user clicks on this button, the application exits. The pane below this is a Label.

The third pane is a Dialog containing a Label "I am a dialog form"; an AsciiString text field with an initial value; and a Command button marked "ok". The next pane is a Box containing a Label marked "label" and a Command button marked "command". The Clock and Load widgets form the next two panes.

The seventh pane is another Box with an AsciiString widget and a Scrollbar. The last pane in figure 3 is an AsciiFile widget displaying */etc/motd*.

The application that instantiated this entire hierarchy is less than 200 lines of C source, including four callback procedures.

Creating New Widgets

Small changes to existing widgets can be made through the general facilities of the resource and translation managers. With these managers, one can make changes to color, font, key bindings, and the such. Many changes that would require a new interface object in other toolkits can be done through resource changes to the existing widgets in the X Toolkit.

The simplest case of creating a new widget is to create a subclass of an existing widget. For example, if one wanted a command button with two lines of text, s/he might create a subclass of Command. Since all superclass structures are inherited, one would merely need to add an additional string field to the instance structure and nothing to the class structure. Since there is no change in user interaction semantics, only the display method would need to be changed. In addition, code to allow the modification of the second text string would be required. All other code could be inherited. This new widget could then be incorporated in any composite widget that allowed buttons.

Subclassing composite widgets is a little more work since new geometry managers may be desired. In general, the additional methods of a Composite widget tend to be complex and the new subclass will want to inherit as much as possible.

One potential 'client' of a widget that the implementor of the widget should keep in mind is the writer of the next widget. With a little thought, the designer of a new widget can decide which of the new methods

added by the subclass should be declared as class variables, which as widget instance variables and which as in-line code.

Class variables are the best way to export methods to potential future subclasses. By installing a procedure pointer into a class variable, the new subclass has the option to easily inherit the old method or to define its own implementation.

The developer of a family of widgets may find circumstances in which a small modification to a parent class would make a new subclass much easier to implement. From the application point-of-view, widget instance and widget class data structures are opaque types. The effect of an addition to one of these structures can be isolated to subclasses of the changed widget without breaking existing application code.

A subclass is free to manipulate any of the widget instance variables defined by its superclasses. The subclass must be careful if it simultaneously manipulates superclass instance data and inherits superclass methods that use the same data. At present, the only way to determine such side-effects is by examination of the superclass implementation(s). We hope to establish conventions in the future for documenting all the public and semi-public interfaces of a widget.

Widgets Of The Future

On our wish list are:

read-only dial: posts a position between 0 and 1 to a circular dial. Useful for indicating position or state.

title bar: the mandatory title, horizontal lines, and close button. A convenience widget, like Dialog.

menus: pull-down, pull-aside, and deck-of-cards menus.

index pane: allows the user to select from a scrollable list of text strings. Useful for selecting topics or filenames.

video label: handles the display of video from an external source inside a widget.

video command button: extends the command button to handle video windows.

text sinks: a variety of additional Text sinks is desirable. One such possibility is a sink that interprets bytes from the Text source as ANSI Terminal control sequences. Another desirable sink would interpret font change control sequences in the source and display multi-font text.

viewport (frame): a movable (scrollable) window frame on a larger widget.

data plotting: a variety of data plotting widgets. Useful for engineering curriculum applications.

Many of these widgets are required by the visual courseware projects at Project Athena and will be implemented over the next several months.

Issues

There are several open issues in the effective design and implementation of new widget classes:

- The class mechanism has only a simple inheritance structure. Therefore, a widget class gets both its display and interaction semantics from its single parent class. This leads to an awkwardness and duplicate code in composing widgets where multiple inheritance is really desired. One way around this is to export little procedures that may be used by other widgets to minimize duplicate code, but this is clearly undesirable.
- One example where simple inheritance influenced our widget implementation is the Text widget. Our preferred hierarchy would be a simple Text class, implementing only the editing and selection actions (without scrolling options) and to implement ScrolledText as a Form consisting of a Scrollbar, optionally some Command buttons, and the Text primitive.
- This would give the user interface designer the maximum flexibility: by customizing the ScrolledTextForm, taking advantage of the constraint-based geometry manager of Form, scrolling could be controlled by a variety of interaction devices.
- It is desirable, however, for ScrolledText to have the semantics of a Text widget from the programmer's viewpoint, and so ScrolledText cannot be implemented as a trivial Form instance without knowing the internals of Text.
- The Athena widgets have been implemented without the aid of a graphic artist. We hope that the implementations and interfaces are amenable to 'dropping in' new display methods should the opportunity arise. As presented now, the visual appearance can be reasonably characterized as stark.
- An infinite variety of convenience widgets that are fixed composites of other widgets (like Dialog and Titlebar) are possible. Before implementors go too wild defining such things and adding them to the standard library, we need to implement an example of a widget that determines its content entirely through the Resource Manager. Then, when there is a resource editor, new composite forms can be built using non-programmatic methods and the widget library can be trimmed back to just the primitives.
- Whether composite widgets are separately written as new classes or built from a resource record by a 'ConfiguredForm' widget, there is the issue of how much access a programmer should have to the individual component widgets. Good programming practice tells us that even composite widgets should always be opaque. Realism tells us that programmers will always want to look behind the curtain.
- As will no doubt have become clear to the reader by this point, the Xtk does not (yet) aspire to be a full

UIMS, for reasons discussed at the opening. It is our desire to see the Xtk as the lower level of a UIMS.

Summary

The M.I.T. X Toolkit, comprised of both general-purpose Intrinsics for widget construction and implementations of widgets for direct application use, provides highly extensible mechanisms for application user interface construction.

The X Toolkit layer sits above Xlib but below a full UIMS layer. By providing a consistent application programming interface to widgets, the Xtk allows independent development of an application and its associated user interface.

Each of the widgets in the Athena widget set leaves as much as possible configurable by the end-user of an application, while still providing reasonable and useful defaults. We expect that, through use, this basic widget set will be extended and improved.

All of the work described here, including complete source code, will be released by M.I.T. with the next release of the X Window System. The Massachusetts Institute of Technology, Digital Equipment Corporation, and others maintain copyright protection over all materials released by Project Athena with explicit permission granted for unlimited use, modification and re-distribution for any purpose and without fee.

Acknowledgments

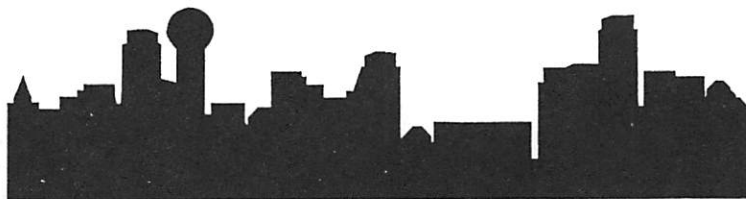
Many individuals have contributed to the design and implementation of the Intrinsics and of the Athena Widgets. To name them all is impossible. Special mention must, however, go to Charles Haynes, Phil Karlton, Tom Benson, Mike Gancarz, Harry Hersh, Mike Chow, Loretta Guarino-Reid, Rich Hyde, Kathy Langone, Mary Larson, Joel McCormack, Leo Treggiari, Jake VanNoy, Terry Weissman, Smokey Wallace, Susan Angebranndt, Ram Rao, Chuck Price, Al Mento, Peter Smith, Jeanne Rich (all from DEC); Frank Hall, Tom Houser, Ed Lee, Rick McKay, Fred Taft, Ted Wilson, Phil Gust (all from H-P); Jack Palevich (formerly H-P); John Osterhaut (U.C. Berkeley); Steve Pitschke (Stellar), Richard Carling (Masscomp); Ron Newman, Dan Geer, Bill Cattey, Russ Sasnett, Steve Wertheim, Stewart Hou, Chris Peterson (M.I.T. Athena); and Matt Hodges (DEC and M.I.T. Athena). Each of the above has made significant contribution to the X Toolkit distribution.

References

1. Anon., *X Window System Protocol*, M.I.T. Software Distribution Center (September, 1987).
2. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions On Graphics* 5(2) pp. 79-109 (April, 1987).
3. J. Gettys, R. Newman, and R. W. Scheifler, *Xlib*

- *C Language Interface*.

4. R. Rao and S. Wallace, "The X Toolkit," in *Usenix Conference Proceedings*, , D.E.C. Western Software Laboratory (Summer, 1987).
5. Anon., *X Toolkit Intrinsics*, M.I.T. Software Distribution Center (September, 1987).



David Rosenthal
Sun Microsystems¹
2550 Garcia Ave.
Mountain View CA 94043
dshr@sun.com

A Simple X11 Client Program -or- How hard can it really be to write “Hello, World”?

ABSTRACT

The “Hello, World” program has achieved the status of a koan in the UNIX community. Various versions of this koan for Version 11 of the X Window System are examined, as a guide to writing correct programs, and as an illustration of the importance of toolkits in X11 programming.

‘... fourth graders discuss whether the machines prefer running simpler programs (“It’s easier for them,” “They hardly have to do any work.”) or more complicated ones (“They feel proud,” “It’s like they are showing what they can do.”).’

Sherry Turkle, *The Second Self*.

Introduction

The “Hello, World” program has achieved the status of a koan in the UNIX community. The spare elegance of:

```
#include <stdio.h>

main()
{
    printf("Hello, World\n");
}
```

has been much admired, and contrasted with the baroque complexity of other system’s attempts to solve the problem.²

Various versions of the “Hello, World” koan for Version 11 of the X Window System are examined. They provide a guide to writing correct programs for X11, insight into the complexity of the issues they have to deal with, and an illustration of the importance of toolkits in X11 programming.

Specification

The “Hello, World” problem needs to be restated slightly for the world of window systems. The program needs to:

- Create a window of an appropriate size, and position it on the screen if required.
- Paint the string “Hello, World” in a suitable font, centered in the window.
- Paint the string in a color which will contrast with the window background – it is not acceptable to paint an invisible string.
- Deal with its window being exposed – repainting the window if required.
- Deal with its window being resized – re-centering the text.
- Deal with its window being closed into an icon, and re-opened, providing suitable identification of the icon.

Note that this specification is much more complex than the canonical “Hello, World” program, so that it is likely that the resulting programs will be significantly more complex.

Vanilla X11

The first “Hello, World” example uses only the facilities of the basic X11 library.

Program Outline

- Open a connection to the X server. Exit gracefully if you cannot.
- Open the font to be used, and obtain the font data to allow string widths and heights to be calculated.
- Select pixel values for the window border, the window background, and the foreground that will be used to paint the characters.

¹Copyright © 1987 by Sun Microsystems, Inc. Permission to use, copy, modify and distribute this document for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies, and that the name of Sun Microsystems, Inc. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Sun Microsystems, Inc. makes no representations about the suitability of the software described herein for any purpose. It is provided “as is” without express or implied warranty.

²Of course, it is important to note that this implementation has bugs. A more correct, ANSI C version is shown in the appendix.

- Compute the size and location of the window based on the text string and the font data, and set up the size hints structure.
- Create the window.
- Set up the standard properties for use by the window manager.
- Create a Graphics Context for use when painting the string.
- Select the types of input events that we are interested in receiving.
- Map the window to make it visible.
- Loop forever:
 - Obtain the next event.
 - If the event is the last event of a group of Expose events (that is, it has `count == 0`), repaint the window by:
 - Discovering the current size of the window.
 - Computing the position for the start of the string that will cause it to appear centered in the current window.
 - Clearing the window to the background color.
 - Drawing the string.

Program Text

Please see Program 1 at the end of the paper for a listing of this program.

Does it meet the specification?

- It computes the size of the window from the string and the font, and positions it at the center of the screen.
- It paints the string in the color `WhitePixel()` on a `BlackPixel()` background. It ensures that the appropriate colormap will be used for the window, so that these colors (which may not actually be White and Black) will be distinguishable.
- Every time it gets the last of a group of Expose events, it enquires the size of the window, and paints the string centered in this space. In particular, it will get an Expose event initially as a consequence of its mapping the window, and will thus paint the window for the first time.
- The same mechanism copes with part or all of the window being exposed. The program will re-paint the entire window when any part is exposed; in this case the effort of only repainting the exposed parts is excessive.
- The fact that the string is re-centered every time the window is painted means that the program deals with re-sizing correctly. Subject to the caveats below, when the window is resized, an Expose event will be generated, and the window will be re-painted.
- The standard properties that the program sets include a specification of a string that a window

manager can display in the icon. The program sets this to be the string it is displaying, so it copes with being iconified. When it is opened, an Expose event will be generated, and the window will be re-painted.

Design Issues

Although this implementation of "Hello, World" is alarmingly long, it is structurally simple. Nevertheless, there are many detailed design issues that arise when writing it. This section covers them, in no particular order.

Repaint Strategy.

Every X11 application has the responsibility for re-painting its image whenever the server requests it to. It is possible to refresh only the parts requested, or to refresh the entire window. The "Hello, World" image is simple enough that refreshing the entire image is a sensible approach.

Exposing part or all the window results in the server painting the exposed areas in the window background color, and one or more Expose events. Each carries, in the `count` field, the number of events in the same group that follow it. After receiving the last of each group, identified by a zero `count`, the window is re-painted.

Re-painting on *every* Expose event would result in unnecessary multiple repaints. For example, consider a "Hello, World" that appears for the first time with one corner overlain by another window. The newly exposed area consists of two rectangles, so there will be two Expose events in the initial group.

We actually take even more rigorous measures to avoid multiple repaints. Every time we decide to repaint the window, we scan the event queue and remove all Expose events that have arrived at the client, but which have yet to arrive at the head of the queue.

Resize Strategy

The first time the window is painted, it seems as if enquiring the size of the window is unnecessary. We have, after all, just created the window and told it what size to be. But X11 does not allow us to assume that the window will actually get created at the requested size; we have to be prepared for a window manager to have intervened and overridden our choice of size. So it is necessary to enquire the window size on the initial Expose event.

When the window is resized, the client needs to re-compute the centering of the text. The implementation does this on the last of every group of Expose events. This raises two questions

- Does every resize of the window result in at least one Expose event?

Consider a window, not obscured by others, that is resized to make it smaller. The X11 server actually has enough pixels to fill the new window size; there is no need to generate an Expose event to

cause pixels to be repainted. This is the simplest example of what the X11 specification calls “Bit Gravity”. Clients may reduce the number of Expose events they receive by specifying an appropriate Bit Gravity. Even if the window is made larger, the Bit Gravity can tell the server how to re-locate the old pixels in the new window to avoid Expose events on parts of the window whose contents are not supposed to change.

By default, X11 sets the Bit Gravity of windows to **ForgetGravity**. This ensures that the gravity mechanism is disabled, Expose events occur on all resizings of the window, and the “Hello, World” program operates correctly if the whole issue of Bit Gravity is ignored. In this default case, the answer to the question is “Yes, every resize results in at least one Expose event”.

But we can exploit Bit Gravity to avoid unnecessary repaints by setting it to **CenterGravity**. This will preserve the centering of the text if the window is resized smaller without involving the client program. In this case, the answer to the question is “No, not every resize results in an Expose event”. But in the cases where no Expose event occurs, the window will still be correct.

- Can we avoid the overhead of enquiring the window size on every re-paint?

As we have seen, Expose events have no direct relation to window re-sizing. In general, X11 clients should listen for both:

- Expose events, which tell them to re-paint some part of the window.
- ConfigureNotify events, which tell them that the window has been changed in some way which requires that the image be re-computed. These carry the new size of the window, there is no need for an explicit enquiry.

In principle, “Hello, World” should only re-compute the centering of the text when it gets a ConfigureNotify event. But the overhead of the extra round-trip to the server to enquire about the size of the window on every Expose event is not critical for this application, and the code in this case is much simpler.

Communicating with the Window Manager

Every X application must use some properties on its window to communicate with the window manager.

- The WM hints, containing information for the window manager about the input and icon behavior of the window. In the case of “Hello, World”, this information is known at compile time and can be initialized statically.
- The size hints, containing information about the size and position of the window. These cannot be initialized statically since they depend on the font properties, and are only known at run-time.

- Other properties, including **WM_NAME**, **WM_ICON_NAME**, and **WM_COMMAND**, which are used to communicate strings such as the name of the program running behind the window.
- The colormap field of the window is not a property, but it may also be used to communicate with the window manager. The conventions about this have yet to be fully specified, and the topic is covered in the next section.

Pixel Values and Colormaps

To ensure “Hello, World” works on both monochrome and color displays, we use the colors Black and White³. To paint in a color using X11, you need to know the pixel value corresponding to it; the pixel value is the number you write into the pixel to cause that color to show on the screen.

Although X11 specifies that Graphics Contexts are by default created using 0 and 1 for the background and foreground pixel values, an application cannot predict the colors that these pixel values will resolve to. It cannot even predict that these two colors will be different, so every application must explicitly set the pixel values it will use.

Pixel values are determined relative to a Colormap; X11 supports an arbitrary number of colormaps, with one or more being installed in the hardware at any one time. X11 supports these colormaps even on monochrome displays. There is a default colormap, which applications with modest color requirements are urged to use, and “Hello, World” is as modest as you could wish. In fact, the colors Black and White have pre-defined pixel values in the default colormap, and we can use these directly.

However, using the pre-defined values means that “Hello, World” becomes dependent on having the default colormap installed. Unless it is, they may not be distinguishable. Unfortunately, when a window is created using **XCreateSimpleWindow()** the colormap is inherited from the parent (in our case, the root window for the default screen), and it is possible that some client may have set the colormap of the root to something other than the default colormap. So, for now, we have to set the colormap field of the window explicitly to be the default colormap, although it is anticipated that when the conventions for window management are finally determined this code will be unnecessary.

Simply setting the colormap field of the window does not ensure that the correct colormap will actually be installed. The whole question of whether, and when, clients should install their colormaps is open to debate at present. There are two basic positions:

- Clients should explicitly install their own colormap when appropriate, for example when they obtain the input focus.

³These colors may not actually be black and white, but they are guaranteed to contrast with each other.

This has two disadvantages, in that it makes every client much more complicated (it means, for example, that “Hello, World” has to worry about the input focus!), and that it means that every client will be doing the wrong thing eventually, when window managers start doing the right thing (whatever that is!). It does, however, mean that clients will work right now.

- Clients should never install their own colormaps, and should assume that some combination of the internals of the server, and the window manager, will do it for them.

This has the disadvantage that it will not work at present, since existing window managers don’t appear to do anything with colormaps.

Strictly speaking, therefore, “Hello, World” should deal with installing colormaps, since the policy has yet to be determined. But it would make the code so complex as to be out of the question for a paper such as this.

Error Handling

It appears that this “Hello, World” implementation follows the canonical “Hello, World” implementation in the great UNIX tradition of optimism, by ignoring the possibility of errors. Not so, the question of error handling has been fully considered:

- On the **XOpenDisplay()** call, we check for the error return, and exit gracefully.
- When opening the font, we cannot be sure that the server will map that name into a font. So we check the error return, and exit gracefully, if the server objects to the name.
- For all other errors, we depend on the default error handling mechanism. When an X11 client gets an Error event from the server, the library code invokes an error handler. The client is free to override the default one, which prints an informative message and exits, but its behavior is fine for “Hello, World”.

Of course, one might ask why we need to explicitly check for errors on opening the font. Surely, the default error handler does what we want? It is an (alas, undocumented) feature of Xlib that not all errors cause the error handler to be invoked. Some errors, such as failure to open a font, are regarded as failure status returns and are signaled by Status return values – in general any routine that returns Status will need its return tested, because it will have bypassed the error handling mechanism.

Finding a Server.

The particular server to use is identified by the **\$DISPLAY** environment variable³ so it does not need to be specified explicitly. It is a convention among X11 clients that a command-line argument containing a colon is a specification of the server to use, but this version of “Hello, World” does not

support the convention (or any other command-line arguments).

Looping for events.

It is natural to assume that you can write the event wait loop:

```
while (XNextEvent(dpy, &event) {
    . . . . .
}
```

but this is not the case. **XNextEvent()** is defined to be void; it only ever returns when there is an event to return, and errors are handled through the error handling mechanism, rather than being indicated by a return value. So it is necessary to write the event loop:

```
while (1) {
    XNextEvent(dpy, &event);
    . . . . .
}
```

Centering the Text

There are a number of ways to compute the size of the string, in order to center it in the window. The most correct method is to use **XTextExtents()**,

Table 1: Aggregate Statistics - Vanilla	
Statistic	Value
number of writes	4
number of reads	12
bytes written	392
bytes read	1736
number of requests	16
number of errors	0
number of events	1
number of replies	3

which computes not merely the width of the string, but also the maximum and minimum Y value for the characters in the string. The example uses **XTextWidth()** to compute the width of the string, and uses the maximum and minimum Y values over all characters in the font. It is to some extent a matter of taste which looks better, the string “Hello, World” has no descenders so the example will tend to locate it somewhat higher in the window than is visually correct.

Protocol Usage

Benchmarking X11 applications is an interesting problem. The performance as seen by the user is affected both by the client, and by the server. To measure the performance of clients independently of any server, I have instrumented the X11 library to gather statistics on the usage of the protocol. The results for the vanilla X11 “Hello, World” program are shown in Tables 1 (aggregate statistics) and 2 (usage of individual requests).

In interpreting these tables, the important thing to remember is that a round-trip to the server (a request that needs a reply) is relatively expensive. For

³Non-UNIX systems will use some other technique.

example, the “replies” entry in Table 1 shows that there were 3 round-trips, and in Table 2 they can be identified as being the GetWindowAttributes, GetGeometry, and QueryFont requests.

This brings out an interesting point. Where did the GetGeometry request come from? The answer is

Table 2: Usage of Requests - Vanilla	
Request	Count
CreateWindow	1
ChangeWindowAttributes	2
GetWindowAttributes	1
MapWindow	1
GetGeometry	1
ChangeProperty	5
OpenFont	1
QueryFont	1
CreateGC	1
ClearArea	1
PolyText8	1

that the `XGetWindowAttributes()` X11 library call uses both the `GetWindowAttributes` and `GetGeometry` protocol request. It is easy to assume that there is a one-to-one mapping between X11 library calls and protocol requests, but this is not the case. The use of `XGetWindowAttributes()` in this “Hello, World” program is inefficient, `XGetGeometry()` should be used instead.

Defaults

This program wires-in a large number of parameters, through the following statements:

```
if((fontstruct = XLoadQueryFont(dpy, FONT))==NULL)
{
    bd = WhitePixel(dpy, DefaultScreen(dpy));
    bg = BlackPixel(dpy, DefaultScreen(dpy));
    fg = WhitePixel(dpy, DefaultScreen(dpy));
    pad = BORDER;
    bw = 1;

    xsh.height = fth + pad * 2;
    xsh.width = XTextWidth(fontstruct, STRING,
                          strlen(STRING)) + pad * 2;
    xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy))
            - xsh.width) / 2;
    xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy))
            - xsh.height) / 2;
```

Ideally, and certainly for any real application, the user should be able to override these wired-in defaults. X11 supplies a default database mechanism to address this problem.

Program

Here is the first example, modified to use the X11 default database mechanism to allow the user to specify values for all these defaults. For each of the first group, it uses `XGetDefault()` to obtain a string value, and then parses it (using `XParseColor()`, or `atoi()`) to the required value. In the case of the

window geometry values in the second group, X11 provides a single mechanism (`XGeometry()`) to parse a string into some or all of the parameters specifying the geometry of a window. To save space, and because the changes to deal with defaults are restricted to a small part of the code, they are presented as a context diff at the end of the paper.

Protocol Usage

Turning statistics gathering on for this version, we get Tables 3 and 4. The extra replies come from the mappings between string names and colors for the foreground, background, and border. These mappings could have been done with a single `AllocNamedColor` request each, instead of a `LookupColor/AllocColor` pair, but this would not have supported the convention that colors can be

Table 3: Aggregate Statistics - Defaults	
Statistic	Value
number of writes	10
number of reads	24
bytes written	504
bytes read	1928
number of requests	22
number of errors	0
number of events	1
number of replies	9

Table 4: Usage of Requests - Defaults	
Request	Count
CreateWindow	1
ChangeWindowAttributes	2
GetWindowAttributes	1
MapWindow	1
GetGeometry	1
ChangeProperty	5
OpenFont	1
QueryFont	1
CreateGC	1
ClearArea	1
PolyText8	1
AllocColor	3
LookupColor	3

specified by strings like “#3a7”.

The Toolkit

Examining the preceding examples, anyone would admit that the basic X11 library fails the “Hello, World” test. Even the simplest “Hello, World” client takes 40 executable statements, and 25 calls through the X11 library interface.

All is not lost, however. It was never intended that normal applications programmers would use the

basic X11 library interface. An analogy is that very few UNIX programmers use the raw system call interface, they almost all use the higher-level "Standard I/O Library" interface. The canonical "Hello, World" program is an example.

The X11 distribution includes a user interface toolkit, intended to provide a more congenial environment for applications development in exactly the same way that *stdio* does for vanilla UNIX. Using this toolkit, the following example shows that X11 can pass the "Hello, World" test with ease.

Program Outline

The outline of the program is:

- Create the top level Widget that represents the toolkit's view of the (top-level) window.
- Create a Label Widget to display the string, overriding the defaults database to set the Label's value to the string to display.
- Tell the top level Widget to display the label, by adding it to the top level Widget's managed list.
- Realize the top level Widget (and therefore its sub-Widgets). This process creates an X11 window for each Widget, setting its attributes from the data in the Widget.
- Loop forever, processing the events that appear.

Program Text

Please see Program 2 at the end of the paper.

Does it meet the specification?

This implementation of "Hello, World" fulfills the specification:

- The window is sized as a result of the geometry negotiation between the top level Widget and its sub-Widgets (in this case the label), so that by default the window is sized to fit the text.
- The default attributes for the Label Widget specify that the text is centered, and the default mechanism supplies a suitable font.
- In the same way, the default mechanism supplies background and foreground colors for the Widget.
- The toolkit manages all Expose events, routing them to appropriate Widgets. Thus, the program behaves correctly for exposure.
- The Label Widget recomputes the centering of the text whenever it is being painted, so that resizing is handled correctly.
- The toolkit handles communicating with the window manager about icon properties, so that iconification is handled correctly.

Design Issues

Note that this implementation isn't merely much shorter than the earlier examples. It has an additional useful feature, in that any or all the values from the default database used by the program can be overridden by command line arguments. **XtInitialize()** parses the command line and merges any

specifiers it finds there with the defaults database.

The toolkit also provides peace of mind by organizing the error handling correctly. Although the documentation of error handling in the toolkit manual is sparse, experiments seem to show that the implementation is satisfactory, providing intelligible messages and sensible behavior.

Protocol Usage

Turning statistics gathering on for the toolkit

Table 5: Aggregate Statistics - Toolkit	
Statistic	Value
number of writes	11
number of reads	25
bytes written	832
bytes read	2012
number of requests	29
number of errors	0
number of events	4
number of replies	9

Table 6: Usage of Requests - Toolkit	
Request	Count
CreateWindow	2
MapWindow	1
MapSubwindows	1
ConfigureWindow	1
InternAtom	2
ChangeProperty	5
OpenFont	1
QueryFont	1
CreatePixmap	3
CreateGC	3
FreeGC	1
PutImage	1
PolyText8	1
AllocColor	3
LookupColor	3

version gives Tables 5 and 6.

These tables reveal that:

- Use of the toolkit does not result in significantly greater protocol traffic.
- The toolkit does not use GetWindowAttributes or GetGeometry. Its repaint and resize strategies use the information in Expose and ConfigureNotify events, and don't require round-trips.

To compare the performance of the toolkit and vanilla versions, we can contrast the cost of being resized by the *uwm* window manager. This is more meaningful than the total cost since startup, since in general window system clients are well advised to invest extra effort in startup code to improve response

Table 7: Cost of resize with <i>uwm</i>		
Item	Vanilla	Toolkit
number of writes	6	4
number of reads	15	4
bytes written	128	112
bytes read	248	160
number of requests	8	4
number of events	3	5
number of replies	4	0
GetWindowAttributes	2	0
GetGeometry	2	0
ConfigureWindow	0	1
ClearArea	2	0
PolyText8	2	3

to interactions.

Table 7 shows that the simplistic repaint and resize strategies of the Vanilla version cost significantly more in terms of the number of round-trips (4 vs. 0) and the amount of data transferred (344 vs. 272).⁴

Conclusions

These examples demonstrate that programming applications using only the basic X library interface is even more difficult and unrewarding than programming UNIX applications using only the system call interface.

Observing the usage of X11 protocol requests gives a server-independent measure of X11 client performance. This can be a useful tool in debugging and performance-tuning X11 clients, the more so in that the performance of interactive clients is likely to be dominated by the number of round-trips per interaction.

Just as anyone considering developing UNIX applications should use *stdio*, anyone considering developing X11 applications should use a toolkit. There is (at least) one in the distribution, and others are available from other sources. Using a toolkit, you can expect:

- isolation from complex and, as yet, undecided design issues about the interaction between X11 clients and their environment,
- competitive performance, at least in terms of protocol usage,
- and much less verbose and more maintainable source code.

In the case of “Hello, World”, a client that took 40 executable statements to program using the basic X11 library took 5 statements to program using the X11 toolkit. And the toolkit version had more functionality and better repaint performance than a

library version with 60 statements.

Acknowledgements

Richard Johnson posted the first attempt at a “Hello, World” program for X11 to the *xpert* mail list. This, and Ellis Cohen’s praiseworthy attempts to write up the conventions needed for communicating between applications and window managers, inspired me to try to write the “Hello, World” program right.

Jim Gettys, Bob Scheifler, & Mark Opperman all identified bugs and suggested fixes in the Vanilla version. The toolkit version is derived from *lib/Xtk/clients/xlabel*.

Appendix: ANSI C Hello World

```
#include <stdio.h>

main()
{
    (void) printf("Hello, World\n");
    exit (0);
}
```

- ANSI C specifies that `printf()` returns the number of characters printed.
- It is necessary to `exit()` or `return()` a value from `main()`.

⁴The reasons why resizing with *uwm* results in several events are too complex to go into here (i.e. both *uwm* and the server have bugs).

Program 1

```
#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define STRING    "Hello, world"
#define BORDER    1
#define FONT      "vrb-25"

/*
 * This structure forms the WM_HINTS property of the window,
 * letting the window manager know how to handle this window.
 * See Section 9.1 of the Xlib manual.
 */
XWMHints xwmh = {
    (InputHint|StateHint), /* flags */
    False,                 /* input */
    NormalState,           /* initial_state */
    0,                     /* icon pixmap */
    0,                     /* icon window */
    0, 0,                  /* icon location */
    0,                     /* icon mask */
    0,                     /* Window group */
};

main(argc,argv)
    int argc;
    char **argv;
{
    Display      *dpy;          /* X server connection */
    Window        win;          /* Window ID */
    GC            gc;           /* GC to draw with */
    XFontStruct   *fontstruct; /* Font descriptor */
    unsigned long fth, pad; /* Font size parameters */
    unsigned long fg, bg, bd; /* Pixel values */
    unsigned long bw;           /* Border width */
    XGCValues      gcval;        /* Struct for creating GC */
    XEvent         event;        /* Event received */
    XSizeHints     xsh;          /* Size hints for window manager */
    char          *geomSpec;     /* Window geometry string */
    XSetWindowAttributes xswa; /* Temporary Set Window Attribute struct */

    /*
     * Open the display using the $DISPLAY environment variable to locate
     * the X server. See Section 2.1.
     */
    if ((dpy = XOpenDisplay(NULL)) == NULL) {
        fprintf(stderr, "%s: can't open %s\n", argv[0], XDisplayName(NULL));
        exit(1);
    }

    /*
     * Load the font to use. See Sections 10.2 & 6.5.1
     */
    if ((fontstruct = XLoadQueryFont(dpy, FONT)) == NULL) {
        fprintf(stderr, "%s: display %s doesn't know font %s\n",
            argv[0], DisplayString(dpy), FONT);
        exit(1);
    }
    fth = fontstruct->max_bounds.ascent + fontstruct->max_bounds.descent;

    /*
     * Select colors for the border, the window background, and the
     * foreground.
     */
    bd = WhitePixel(dpy, DefaultScreen(dpy));
    bg = BlackPixel(dpy, DefaultScreen(dpy));
    fg = WhitePixel(dpy, DefaultScreen(dpy));

    /*
     * Set the border width of the window, and the gap between the text
     * and the edge of the window, "pad".
     */
}
```

```

pad = BORDER;
bw = 1;

/*
 * Deal with providing the window with an initial position & size.
 * Fill out the XSizeHints struct to inform the window manager. See
 * Sections 9.1.6 & 10.3.
 */
xsh.flags = (PPosition | PSize);
xsh.height = fth + pad * 2;
xsh.width = XTextWidth(fontstruct, STRING, strlen(STRING)) + pad * 2;
xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy)) - xsh.width) / 2;
xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy)) - xsh.height) / 2;

/*
 * Create the Window with the information in the XSizeHints, the
 * border width, and the border & background pixels. See Section 3.3.
 */
win = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
                        xsh.x, xsh.y, xsh.width, xsh.height,
                        bw, bd, bg);

/*
 * Set the standard properties for the window managers. See Section
 * 9.1.
 */
XSetStandardProperties(dpy, win, STRING, STRING, None, argv, argc, &xsh);
XSetWMHints(dpy, win, &xwmh);

/*
 * Ensure that the window's colormap field points to the default
 * colormap, so that the window manager knows the correct colormap to
 * use for the window. See Section 3.2.9. Also, set the window's Bit
 * Gravity to reduce Expose events.
 */
xswa.colormap = DefaultColormap(dpy, DefaultScreen(dpy));
xswa.bit_gravity = CenterGravity;
XChangeWindowAttributes(dpy, win, (CWC colormap | CWBitGravity), &xswa);

/*
 * Create the GC for writing the text. See Section 5.3.
 */
gcv.font = fontstruct->fid;
gcv.foreground = fg;
gcv.background = bg;
gc = XCreateGC(dpy, win, (GCFont | GCForeground | GCBackground), &gcv);

/*
 * Specify the event types we're interested in - only Exposures. See
 * Sections 8.5 & 8.4.5.1
 */
XSelectInput(dpy, win, ExposureMask);

/*
 * Map the window to make it visible. See Section 3.5.
 */
XMapWindow(dpy, win);

/*
 * Loop forever, examining each event.
 */
while (1) {
/*
 * Get the next event
 */
XNextEvent(dpy, &event);

/*
 * On the last of each group of Expose events, repaint the entire
 * window. See Section 8.4.5.1.
 */
if (event.type == Expose && event.xexpose.count == 0) {
    XWindowAttributes xwa; /* Temp Get Window Attribute struct */
    int x, y;

```

```

/*
 * Remove any other pending Expose events from the queue to
 * avoid multiple repaints. See Section 8.7.
 */
while (XCheckTypedEvent(dpy, Expose, &event));

/*
 * Find out how big the window is now, so that we can center
 * the text in it.
 */
if (XGetWindowAttributes(dpy, win, &xwa) == 0)
    break;
x = (xwa.width - XTextWidth(fontstruct, STRING, strlen(STRING))) / 2;
y = (xwa.height + fontstruct->max_bounds.ascent
     - fontstruct->max_bounds.descent) / 2;

/*
 * Fill the window with the background color, and then paint
 * the centered string.
 */
XClearWindow(dpy, win);
XDrawString(dpy, win, gc, x, y, STRING, strlen(STRING));
}
}

exit(1);
}

```


Context diff for second program

```

*** xhw0.c    Sun Dec  6 08:25:11 1987
--- xhw1.c    Sun Dec  6 08:25:12 1987
*****
*** 5,10 ****
--- 5,17 ----
    #define STRING  "Hello, world"
    #define BORDER  1
    #define FONT    "vrb-25"
+ #define      ARG_FONT      "font"
+ #define      ARG_BORDER_COLOR "bordercolor"
+ #define      ARG_FOREGROUND "foreground"
+ #define      ARG_BACKGROUND "background"
+ #define ARG_BORDER      "border"
+ #define      ARG_GEOMETRY  "geometry"
+ #define DEFAULT_GEOMETRY ""

/*
 * This structure forms the WM_HINTS property of the window,
 *****
 *** 29,38 ****
     Display      *dpy;          /* X server connection */
     Window       win;          /* Window ID */
     GC           gc;           /* GC to draw with */
     XFontStruct  *fontstruct; /* Font descriptor */
 !   unsigned long fth, pad; /* Font size parameters */
     unsigned long fg, bg, bd; /* Pixel values */
     unsigned long bw;          /* Border width */
     XGCValues     gcv;          /* Struct for creating GC */
     XEvent        event;        /* Event received */
     XSizeHints    xsh;          /* Size hints for window manager */
--- 36,49 ----
     Display      *dpy;          /* X server connection */
     Window       win;          /* Window ID */
     GC           gc;           /* GC to draw with */
+   char          *fontName; /* Name of font for string */
     XFontStruct  *fontstruct; /* Font descriptor */
 !   unsigned long ftw, fth, pad; /* Font size parameters */
     unsigned long fg, bg, bd; /* Pixel values */
     unsigned long bw;          /* Border width */
+   char          *tempstr; /* Temporary string */
+   XColor         color;    /* Temporary color */
+   Colormap       cmap;     /* Color map to use */
     XGCValues     gcv;          /* Struct for creating GC */
     XEvent        event;        /* Event received */
     XSizeHints    xsh;          /* Size hints for window manager */
 *****
 *** 51,77 ****
     /*
      * Load the font to use.  See Sections 10.2 & 6.5.1
      */
 !   if ((fontstruct = XLoadQueryFont(dpy, FONT)) == NULL) {
 !   fprintf(stderr, "%s: display %s doesn't know font %s\n",
 !       argv[0], DisplayString(dpy), FONT);
 !   exit(1);
 !   }
     fth = fontstruct->max_bounds.ascent + fontstruct->max_bounds.descent;

     /*
      * Select colors for the border, the window background, and the
 !   * foreground.
      */
 !   bd = WhitePixel(dpy, DefaultScreen(dpy));
 !   bg = BlackPixel(dpy, DefaultScreen(dpy));
 !   fg = WhitePixel(dpy, DefaultScreen(dpy));
 !

     /*
      * Set the border width of the window, and the gap between the text
      * and the edge of the window, "pad".
      */
     pad = BORDER;
 !   bw = 1;

```

```

/*
 * Deal with providing the window with an initial position & size.
--- 62,117 ----
/*
 * Load the font to use. See Sections 10.2 & 6.5.1
 */
! if ((fontName = XGetDefault(dpy, argv[0], ARG_FONT)) == NULL) {
! fontName = FONT;
! }
! if ((fontstruct = XLoadQueryFont(dpy, fontName)) == NULL) {
! fprintf(stderr, "%s: display %s doesn't know font %s\n",
! argv[0], DisplayString(dpy), fontName);
! exit(1);
! }
+ fth = fontstruct->max_bounds.ascent + fontstruct->max_bounds.descent;
+ ftw = fontstruct->max_bounds.width;

/*
 * Select colors for the border, the window background, and the
! foreground. We use the default colormap to allocate the colors in.
! See Sections 2.2.1, 5.1.2, & 10.4.
 */
! cmap = DefaultColormap(dpy, DefaultScreen(dpy));
! if ((tempstr = XGetDefault(dpy, argv[0], ARG_BORDER_COLOR)) == NULL ||
! XParseColor(dpy, cmap, tempstr, &color) == 0 ||
! XAllocColor(dpy, cmap, &color) == 0) {
! bd = WhitePixel(dpy, DefaultScreen(dpy));
! }
! else {
! bd = color.pixel;
! }
! if ((tempstr = XGetDefault(dpy, argv[0], ARG_BACKGROUND)) == NULL ||
! XParseColor(dpy, cmap, tempstr, &color) == 0 ||
! XAllocColor(dpy, cmap, &color) == 0) {
! bg = BlackPixel(dpy, DefaultScreen(dpy));
! }
! else {
! bg = color.pixel;
! }
! if ((tempstr = XGetDefault(dpy, argv[0], ARG_FOREGROUND)) == NULL ||
! XParseColor(dpy, cmap, tempstr, &color) == 0 ||
! XAllocColor(dpy, cmap, &color) == 0) {
! fg = WhitePixel(dpy, DefaultScreen(dpy));
! }
! else {
! fg = color.pixel;
! }
/*
 * Set the border width of the window, and the gap between the text
 * and the edge of the window, "pad".
 */
pad = BORDER;
! if ((tempstr = XGetDefault(dpy, argv[0], ARG_BORDER)) == NULL)
! bw = 1;
! else
! bw = atoi(tempstr);

/*
 * Deal with providing the window with an initial position & size.
*****
*** 78,88 ***
 * Fill out the XSizeHints struct to inform the window manager. See
 * Sections 9.1.6 & 10.3.
 */
! xsh.flags = (PPosition | PSize);
! xsh.height = fth + pad * 2;
! xsh.width = XTextWidth(fontstruct, STRING, strlen(STRING)) + pad * 2;
! xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy)) - xsh.width) / 2;
! xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy)) - xsh.height) / 2;

/*
 * Create the Window with the information in the XSizeHints, the
--- 118,150 ----
 * Fill out the XSizeHints struct to inform the window manager. See

```

```

    * Sections 9.1.6 & 10.3.
    */
!   geomSpec = XGetDefault(dpy, argv[0], ARG_GEOMETRY);
!   if (geomSpec == NULL) {
!   /*
!   * The defaults database doesn't contain a specification of the
!   * initial size & position - fit the window to the text and locate
!   * it in the center of the screen.
!   */
!   xsh.flags = (PPosition | PSize);
!   xsh.height = fth + pad * 2;
!   xsh.width = XTextWidth(fontstruct, STRING, strlen(STRING)) + pad * 2;
!   xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy)) - xsh.width) / 2;
!   xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy)) - xsh.height) / 2;
!   }
!   else {
!   int      bitmask;
!
!   bzero(&xsh, sizeof(xsh));
!   bitmask = XGeometry(dpy, DefaultScreen(dpy), geomSpec, DEFAULT_GEOMETRY,
!   bw, ftw, fth, pad, pad, &(xsh.x), &(xsh.y),
!   &(xsh.width), &(xsh.height));
!   if (bitmask & (XValue | YValue)) {
!       xsh.flags |= USPosition;
!   }
!   if (bitmask & (WidthValue | HeightValue)) {
!       xsh.flags |= USSize;
!   }
!   }

!   /*
!   * Create the Window with the information in the XSizeHints, the

```

Program 2

```
#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>
#include <X11/Atoms.h>
#include <X11/Label.h>

#define STRING "Hello, World"

Arg wargs[] = {
    XtNlabel, (XtArgVal) STRING,
};

main(argc, argv)
    int argc;
    char **argv;
{
    Widget      toplevel, label;

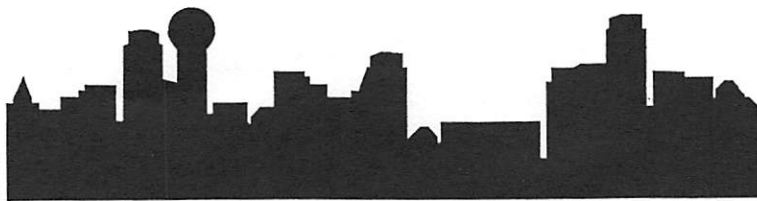
    /*
     * Create the Widget that represents the window.
     * See Section 14 of the Toolkit manual.
     */
    toplevel = XtInitialize(argv[0], "XLabel", NULL, 0, &argc, argv);

    /*
     * Create a Widget to display the string, using wargs to set
     * the string as its value. See Section 9.1.
     */
    label = XtCreateWidget(argv[0], labelWidgetClass,
                           toplevel, wargs, XtNumber(wargs));

    /*
     * Tell the toplevel widget to display the label. See Section 13.5.2.
     */
    XtManageChild(label);

    /*
     * Create the windows, and set their attributes according
     * to the Widget data. See Section 9.2.
     */
    XtRealizeWidget(toplevel);

    /*
     * Now process the events. See Section 16.6.2.
     */
    XtMainLoop();
}
```

USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

The Counterpoint Fast File System

Dr. J. Kent Peacock
Counterpoint Computers
San Jose, CA 95131
408-434-0190
hplabs!oblio!kent

ABSTRACT

The System V File System implementation suffers from a severe performance problem in sequential access to files. Counterpoint's solution to this problem is to move multiple contiguous blocks of files between disk and the file system buffer cache with a single I/O operation. With this technique, we can achieve larger transfer sizes than those afforded by simply using a larger block size in the file system. In order to take advantage of this, we have changed the free list organization to a bit map, which allows the allocation of contiguous ranges of blocks to files. We have also made additional enhancements to the file system to take advantage of Counterpoint's multi-processor architecture by allowing parallel file system operations on different processors. In the course of tuning the write performance of the file system, we found an interesting bottleneck and solved it by a modification to the disk queue sorting algorithm. Lastly, we made performance measurements indicating best-case performance which is very good both in absolute terms and relative to more expensive technology.

Introduction

The System V File System suffers from a significant performance disadvantage with respect to the 4.2 and 4.3 versions of Berkeley UNIX. Because of this, UNIX vendors who base their system implementations on System V are forced to enhance their file systems in order to be competitive in the marketplace against Berkeley-based systems. Many vendors accomplish this enhancement by transplanting the Berkeley file system into their System V systems. Counterpoint solved this problem by modifying the System V file system to provide higher throughput with minimum change to the underlying file system structure and semantics.

The Problem with System V

The measure of throughput that we are most interested in is the rate at which bytes of a UNIX file can be read or written sequentially, since this is the dominant mode of access in most UNIX systems. The random access mode of access is less interesting to us, since the throughput rate tends to be limited by disk access time. The main solution to random access performance inadequacy is faster-seeking disks or more disk arms. Software improvements to random access throughput, such as seek optimization, are already incorporated in System V. On the other hand, very little is done in System V to use the highly correlated and predictable pattern of sequential access to increase throughput. One performance enhancement is the reading ahead of the next block in a file when

sequential access is detected, so that application processing of the current block and the disk operation for the succeeding block can be overlapped.

Another very important performance feature of the UNIX file system is the disk buffer cache in system RAM. This cache allows access to a set of recently-used disk blocks without I/O. The effect of this cache is very dramatic, and properly tuned systems often run with a high cache hit rate and almost no disk I/O. This is especially true of applications, such as the C compiler, which consist of a set of steps, each of which produces temporary files to be read by the following step. These files are written into the cache when created, and usually are still in the cache when read, so that practically no I/O operations are done. In fact, since each temporary file is removed after reading, it is possible for blocks in a file never to be written to the disk during the lifetime of the file, even though a physical disk block is allocated to a each logical file block.

System V Free Block List Organization

There is an important property of the UNIX file system that underlies the throughput problem with sequential access. Early versions of UNIX ran on systems with small disks, which motivated the implementors to avoid wasting disk space. To accomplish this, every logical block in a file is explicitly mapped through the file system to a physical block of a disk. Thus, blocks can be allocated individually to a file as they are written and hence there are no physical

blocks allocated to a file that do not contain data. The average wasted space in any sequential UNIX file is therefore one-half of the last block. To avoid overhead when the file system is full, the free list is organized as a list of blocks maintained inside the free blocks themselves. Hence, when there are no free blocks left in a file system, there are also no blocks used to contain any free block information.

Although this UNIX free list organization wastes little space, it has a very serious drawback. The free list is maintained as a linked set of free list blocks, each of which contains a list of free block numbers. The system keeps the block numbers from the first free list block in memory for fast block allocation and freeing. When the memory resident list is exhausted during allocation, the next free list block is read from disk. Similarly, when the memory list is filled due to block freeing, the list is written out to the current free list block and a new free list block is put on the head of the list. Since the system has access at any given time to around the first 50 free block numbers on the disk, it is very difficult for the free list to be maintained in any semblance of sorted order, so it isn't. What this means in terms of locality of blocks allocated to each file is that there is generally none. The logical blocks of a given file are likely to be mapped to physical blocks scattered randomly throughout the disk. The sequential access performance is the same (except for read-ahead) as the random access performance! Matters are not always this grim, however. File systems are created initially with free lists that are sorted and interleaved according to a rotational gap parameter. Files allocated on a fresh file system are thus allocated with almost optimal locality, to the point of having the correct rotational gap between blocks to avoid incurring an entire disk revolution between blocks. Unfortunately, the dynamic allocation and freeing of blocks in a file system causes the ordering to degrade over time to randomness, particularly since blocks are freed and reallocated in Last-In-First-Out order. In fact, to compensate for this, blocks in a file are freed from the end backwards when a file is removed. It should also be noted that the file system checking program, *fsck*, can also re-sort the free list in an old file system. This is useful except that free blocks tend to be widely scattered in "mature" file systems, so that the free list is not likely to be restored to its original near-optimal performance characteristics.

Where does this leave us in terms of unmodified System V performance on Counterpoint hardware? Using our original ST506 disk controller design, we could process almost two blocks per revolution of a 3600 RPM disk on a newly created file system, for a throughput of about 100 Kbytes/second. With a randomized free list, the throughput dropped to less than one block per revolution or 50-60 Kbytes/second. On the current SCSI controller, we obtain at best one block per revolution. Some drive types with inefficient controllers can transfer less than one block per revolution, or around 30-40 Kbytes/second. We

show dramatic improvement over these measurements with the modified file system. We should note that once a file is fully resident in the disk buffer cache, it can be read at a rate of 800-900 Kbytes/second, this limit being due to processor saturation.

Bit Map Free List Organization

There is a widely-used free list mechanism which is superior to the System V free list structure in almost every respect, namely *bit map* free lists. A bit map free list is simply an array of bits such that if bit *i* is 1, then block *i* is free; otherwise block *i* is allocated. The primary advantage of this technique is that the free block map is implicitly sorted, and blocks which are close on the disk are also close in the bit map. The fact that the allocation state of sets of blocks in close proximity can be accessed easily means that the allocation policy can be completely arbitrary, including allocation of groups of blocks to a file or arbitrarily complex rotational and seek latency optimizations. A small drawback of bit maps is that they require disk space proportional to the file system size to store the disk resident image of the free block list. Of more concern is the possible need to load the entire bit map into system memory for each file system. A second disadvantage is that allocation time increases when the number of free blocks is very small, due to the need to scan a large portion of the bit map to find the free blocks. Although neither of these problems is very serious, there are simple solutions to both.

The use of some form of bit map free list organization is the cornerstone of any attempt to increase UNIX file system performance. Without this change, the equivalence of sequential and random access performance is almost impossible to break. Indeed, our attempts to improve locality of block allocation to files while retaining the standard free list organization failed, leading us to the unavoidable conversion of the free list to a bit map. One of the puzzling aspects of the history of UNIX releases from AT&T is that such a change has not been incorporated already by them. Code exists in the released versions of *fsck* indicating that at some point a bit map free list was used.

Moving More Bytes per Revolution

To increase disk throughput, we must maximize the time that the disk is actually transferring data, which means minimizing the disk seek and rotational latency and the request processing overhead. Once the blocks of a file can be allocated in close proximity using a bit map free list, the seek time becomes a minor part of the non-transfer time. Therefore, we need to minimize the rotational latency and processing overhead per byte transferred. The processing overhead tends to be fixed, and determines the optimal rotational gap between blocks. The rotational latency depends on how close to the optimal pattern blocks can be allocated.

There are several problems with attempting to

optimize the rotational latency. The first is that it is highly device dependent, so each new device must be characterized and tuned. In the case of Small Computer System Interface (SCSI) devices, which Counterpoint uses, simple characterization of the devices is usually not possible, since they contain embedded controllers. These controllers tend to make the disk response characteristics not easily predictable. Secondly, the processing overhead per request also depends on the application that is reading and processing the data. Add too much processing, and the optimal gap becomes the worst gap because the request slips an extra revolution. Due to the relatively high overhead of SCSI, it may not even be possible to perform more than one request per revolution. Thirdly, the attempted allocation of blocks with optimal spacing complicates the block selection algorithms significantly.

Given that the processing overhead is high, and the optimal rotational gap may be an entire revolution, what should we do to reduce the non-transfer time *per byte*? The simple answer is to transfer more bytes per request. The traditional way to do this has been to increase the file system block size. The earliest versions of UNIX had 512-byte blocks, and one of the favorite local modifications to the system was to increase the blocksize of the file system to 1024 bytes. The current System V implementation allows a few block size choices, the largest of which is 2048 bytes, but does not allow arbitrary specification of the block size without considerable effort. This led to the key feature of the Counterpoint solution to the throughput problem. Instead of increasing the block size to move more bytes per request, we force blocks in files to be contiguous and process more than one file block per I/O request.

The Berkeley File System

We are interested in looking at features of the Berkeley File System because it is a widely-used alternative to System V and because it represents a different approach to that of Counterpoint. The Berkeley file system allows each file system to have a specified block size up to 8 Kbytes, with allocation actually done in terms of blocks and *fragments*, which are partial blocks. The use of a bit map free list allows locality of allocation for blocks, as well as the optimization of rotational latency between allocated blocks. A number of semantic differences exist, including a different directory structure, which make the Berkeley file system not strictly compatible with System V. The cache management and disk allocation routines are also complicated by having to handle various buffer sizes and deal with block fragments. This complication does not exist in System V, nor in Counterpoint's Fast File System. A complete description of the Berkeley File System can be found in [McKusick84].

The Counterpoint Fast File System

The key concept of the Fast File System is the movement of multiple file blocks between the disk buffer cache and the disk using a single I/O request. This is in contrast to the Berkeley file system which gains its performance by moving a single, large block using one I/O request. The Berkeley file system thus moves one block of up to 8 Kbytes at once, whereas the Fast File System allows up to 32 contiguous blocks, or 32 Kbytes, to be moved using one request. (The limit of 32 blocks was selected as a result of tests which indicated that larger transfers gave insignificant performance increases.) Since these 32 blocks are brought into the cache when the first block is requested, the 31 following requests are satisfied at the higher throughput rates afforded by the cache hits.

For this strategy to work, the disk controller hardware must support *scatter-gather* operations, where a single transfer can move data to or from a number of non-contiguous sections of memory. Counterpoint's disk controller has a feature whereby a new transfer address can be loaded at each page boundary crossing. This type of feature is also required to support raw I/O in a paging environment.

Having this ability to move multiple blocks between the disk and the system buffer cache does no good unless the adjacent blocks that are moved have some logical connection. The obvious way to connect these blocks is to make logically adjacent blocks in a file physically adjacent on the disk. In some operating systems, this is done by allocating blocks to files in *extents*, which are groups of contiguous physical blocks that are mapped to groups of contiguous logical blocks. Usually in such allocation schemes, the mapping information translates logical extents to physical extents, rather than mapping individual blocks as in System V. Since one of our objectives was to disturb the file systems structures as little as possible, we chose not to change the block mapping strategy of System V to do extent mapping rather than individual block mapping.

To know when the file system should move multiple blocks, the function which maps a logical block number in a file to its physical block number on disk had to be changed. On read operations, it was modified to determine the range of adjacent logical blocks that map to blocks that are also adjacent physically. On write operations, it is the mapping function which allocates a physical block on the first write to a logical block. This function was modified for write operations to pass the physical address of the previous logical block to the allocation function so that the following physical block could be allocated if it is free. In order to make sure that the physical block requested is available most of the time, the file system block allocator logically divides the file system blocks into groups of *N* adjacent blocks called *clusters*. *N* is a file system parameter, called the *cluster size*, which is typically set to 8 blocks. When a file is

created and the first block is written, the allocator looks for the lowest-numbered free block in the file system, and puts all of the free blocks in that block's cluster into a free block cache associated with the file. Subsequent allocations are satisfied from this small free block cache until it is exhausted. This technique is similar to one used in the DEMOS file system and described in [Powell79]. Each time the file's free block cache becomes empty, the allocator looks for a nearby completely unallocated cluster and puts that cluster's blocks into the file's free block cache. The gap parameter of the file system is used to determine how many clusters to skip before looking for unallocated clusters. This parameter is normally set so that no clusters are skipped in choosing the next available cluster for allocation. When a file is closed or flushed to disk, any blocks left in the free block cache are freed back to the main file system free list. This avoids wasting left-over unused blocks in the last cluster of each file, and also insures that the free list does not lose blocks that are not allocated.

The clustering method and use of the free block cache have a number of important advantages: First, the use of clusters guarantees at least cluster-sized clumps of adjacent blocks in files after the first few blocks. With the gap parameter set as suggested, files actually tend to be allocated as one completely contiguous set of blocks, especially in fresh file systems. Secondly, the use of the file free block cache allows faster allocation when the cache is not empty. Thirdly, the free block cache reduces a contention point when running a multi-processor configuration. Allocation of a block from a file's block cache requires protecting only that file from simultaneous access from more than one processor. On the other hand, allocating blocks from the main free list requires protecting the entire free list from simultaneous access. So with a cluster size of 8 blocks, we need to lock and access the main free list 8 times less often, since we need only do this each time we get a new cluster to put into the file block cache. Fourthly, the allocation of a partial cluster as the first set of blocks in each file helps keep more fully free clusters available. In fact, since most files in UNIX are small, most will consist entirely of a partial cluster. Lastly, since the first available blocks are allocated at the start of each file, and directories are usually small files, this also has the effect of keeping directory blocks close to the beginning of the file system, which improves performance of directory search operations.

An additional form of cacheing is used to improve the read performance of the system. The logical to physical block mapping information associated with each file is organized as a tree structure. The first few blocks of each file can be mapped with no block reads of mapping information, but later blocks require between one and three extra block reads to get to the desired block. By remembering the extra information computed by the *bmap* mapping function about ranges of contiguous blocks in the file, we can compute the physical block number of a logical block

without the extra reads. This is possible when the requested block falls into the previously determined contiguous range. During sequential reads, this is true of all but the first block in each contiguous range.

We noted previously that sequential block read-ahead is beneficial for overlapping I/O with application processing of the data. Because we read more blocks per request with our file system, the delay to completion of an I/O request is correspondingly higher than the normal file system. For this reason, we found the sequential read-ahead of clusters of blocks to be useful in overlapping the larger I/O delay with the larger amount of application processing that is done per I/O request. To accomplish this, *bmap* actually computes two ranges of contiguous blocks, analogous to the read-ahead block supplied by the original System V function.

In System V, there are two block read entry points to the block I/O system, *bread* and *breada*. *Bread* obtains the requested block synchronously, waiting if necessary for I/O completion. *Breada* obtains the requested block synchronously, but also queues an asynchronous request to read the following logical block so that the I/O is overlapped with the processing of the requested block. We have added an additional entry point to the block I/O system, *breadm*, which performs a clustered read on a range of blocks. *Breadm* is passed the two ranges of block numbers computed by the mapping function, and a flag word which indicates whether we are doing sequential access and whether the operation is asynchronous or not. The format of the call to *breadm* is as follows:

```
breadm (dev, bno, behind, ahead,
        flag, nxtstrt, nextend)
```

where *dev* and *bno* are device and block number, *behind* and *ahead* represent the range of contiguous blocks surrounding *bno* computed by *bmap*, *flag* contains flag bits, and *nxtstrt* and *nextend* are the starting and ending block numbers of the next range of contiguous blocks. The function obtains cache buffers for the largest range of non-cached blocks between *behind* and *ahead* which includes *bno*, creates a dummy page table pointing to the buffers obtained, creates an I/O request using a *physio* buffer, and calls the device strategy routine. When the operation completes, a routine called *clustdone* is called which releases all of the buffers other than the one containing *bno* back to the free list. The coding of this function was fairly straightforward due to the ability to make use of the structure already in place to support raw I/O operations. It is interesting that [McKusick84] suggests the possibility of clustering blocks together, but dismisses it as requiring too much change to the disk drivers. We found that almost no change was required in our disk driver, since the implementation could make use of the raw I/O support.

With the *flag* word, it is possible to combine the normal and read-ahead cases into the single *breadm*

function. Read-ahead is accomplished by a recursive call of *breadm* to itself with *bno* = *behind* = *nextstrt* and *ahead* = *nextend* and with the asynchronous flag set. The recursive call is triggered when *bno* is in the cache, the sequential flag is set, and the first block of the second contiguous range is not found in the cache. The asynchronous flag tells *breadm* not to recurse, and to release all of the blocks to the free list when the operation completes. In this way, we obtain read-ahead on the scale of entire contiguous ranges of the file.

Absence of the sequential flag is taken to imply that the access pattern is random. When blocks are randomly accessed in a large file, reading up to 32 blocks on each read decreases performance due to the larger delay in completing the request. Therefore, non-sequential access is limited to a configurable maximum transfer size, normally set to 4 Kbytes.

Most file write activity in UNIX does not result in immediate movement of the data to disk, because the disk cache is organized to use a *write-back* strategy, rather than *write-through*. File system blocks that have been modified are written to disk either during the periodic disk flush (normally caused by the *sync* system call), or when an attempt is made to allocate a modified, unwritten block from the head of the free list. The attempted allocation causes a call to the *bwrite* function, which in this case performs the write operation and returns the block to the free list ready for allocation. As with *bread*, this function causes only one block to be written. We therefore added a function analogous to *breadm*, called *bwritem*, which attempts to bundle contiguous modified blocks in the disk cache into a single write operation to the disk. Since file blocks are allocated in clusters as a file is written, it is very likely that when the first block of each cluster reaches the head of the free list, the other blocks in the cluster will have been written into the cache. Therefore, the search in *bwritem* for blocks adjacent to the one being flushed is almost certain to succeed, so that at least a cluster is written with one I/O operation. The rest of the operation is very similar to that of *breadm*.

An advantage to this whole approach is the relative ease with which the modifications can be made. The original implementation took approximately 3 man months, of which a sizable portion was spent changing file system utilities to understand the bit map free list format. The changes were originally made to a Release 2 kernel, and were incorporated into Release 3 at a later time. The Release 3 integration took much less than 1 man month of a second person's time.

An Interesting Tuning Problem

While tuning the write portion of the file system, an interesting problem was analyzed and solved. We noticed that the utilization of both the processor and disk were on average around 60-70%, an unexpected finding. We expected one or the other of the resources

to be near saturation and limiting the other, given the write-back nature of the cache and the decoupling of the process write activity from the actual I/O operations. On closer examination, it was determined that the two were not completely decoupled. Whenever the file system allocates a new indirect mapping block associated with a file, the writing of the mapping information is done synchronously, that is, the file system waits for the mapping write operations to complete before proceeding further with the data write operation. (This situation occurs after approximately each 256 blocks written to a file.) Now, if the writing process has filled the cache with modified blocks, the cache block allocator forces a large number of asynchronous I/O requests onto the disk queue as it searches for a unmodified blocks to allocate. Along comes the request to allocate an indirect block, which writes the new mapping information to disk synchronously behind the large queue of asynchronous requests. The process goes to sleep for a relatively long time waiting for the synchronous I/O operation, which must in turn wait for most of the asynchronous buffer flush operations. When the writing process awakens, it has fallen behind the rate necessary to keep the disk fed with modified blocks. The process must fill the buffer cache again with modified blocks before the cleaning process starts up again, during which time the disk is idle.

We solved this problem by making a change to the disk request sorting algorithm to sort synchronous requests ahead of asynchronous ones in the disk queue. This allows the process to sleep a much shorter time waiting for the indirect block update, which in turn means that it has a better chance to keep up with the rate required to keep the disk busy. (Note that this does not affect file system consistency, as the disk driver is allowed complete freedom in ordering its queue.) When we made this change, the utilization of the processor and disk increased to 95% and 85%, respectively, and the write throughput increased by 25%.

System V Compatibility

One of our goals in creating the Fast File System was to preserve compatibility with the standard System V File System. The only change that was made to file system structures was the unavoidable conversion of the free list to a bit map structure. As it turns out, this change affects only a handful of administrative utilities, such as *mkfs*, *fsck* and *df*. These changes to the file system structure and the utilities were also done in such a way that file system free list organization can be changed between old and new formats using the *fsck* utility, preserving full backward compatibility. The kernel also remains fully backward compatible with the old file system, allowing mixtures of old and new types to be mounted.

Multiprocessor Considerations

The Counterpoint system is an expandable multi-processor, described in [Peacock87], which has required changes to the kernel to allow effective parallel operation of various system facilities, including the file system. While the general multi-threading of the kernel is beyond the scope of this paper, some comments on the file system multi-threading are appropriate.

The Counterpoint kernel initially supported the so-called *Master-Slave* model, described in [Goble81], in which all system call and trap processing is performed on a dedicated *Master* processor, and the slaves are reserved for running user-level processing only. Over time, through the addition of appropriate multi-processor locks, much of the system level processing has been multi-threaded so that it can be performed on any processor in the system.

In terms of the file system, this means that locks have been added at the *inode* and buffer cache interfaces to allow multi-processor manipulation of the *inode* and cache block resources. Access to the super-block of each file system and to the disk driver itself are still single-threaded through the *Master* processor. These choices are made acceptable by the fact that mechanisms have been added to reduce the number of times the super-block and disk driver are accessed. The super-block allocation activity is lessened by the technique previously described, which caches a cluster of free blocks in the memory-resident *inode* structure. The disk driver is called less frequently due to the clustering of operations provided by the *breadm* and *bwrite* functions. Note that the first read block request of a cluster causes an I/O operation to single-thread through the *Master* which reads the entire cluster (or more) into the buffer cache. Subsequent read operations for the other blocks read in are satisfied from the buffer cache, without having to single-thread.

To measure the degree of contention in multiple processors accessing the buffer cache, a simple test was run. With N processors, N processes were run, each repeatedly reading a different file which was small enough to be contained in the cache. The rates at which each process could read its file were then added together as the aggregate throughput rate of the buffer cache. The following table represents the results obtained:

CPU's	Kbytes/sec
1	891
2	1478
3	2048

Performance

In benchmark comparisons to other systems, we find that the throughput of our SCSI disk implementation compares favorably with systems which use SMD technology and the Berkeley file system. We performed our tests using a Counterpoint System

19K, which benchmarks at about 1.6 times a VAX 11/780 in CPU speed. The disks used were Quantum Q280, CDC Wren III and Fujitsu Eagle drives. The first two disk types connect to the standard SCSI controller provided with System 19K, while the Fujitsu was attached to a Xylogics controller through a VME expansion bus attached to the Counterpoint bus. The sequential read test consists of a single process which repeatedly reads a large file in 16 Kbyte pieces and reports the throughput rate after 10 seconds of reading. (The file was made to be larger than the buffer cache.) In this read test, we can sustain a rate of 450-550 Kbytes per second, depending on the particular disk used. This is in contrast to a normal System V file system with the same disks. Running the same test on a VAX 11/750 Berkeley 4.2 system with CDC SMD disks on a UNIBUS controller, we found we could achieve 280 Kbytes per second. We attribute the higher performance to the Fast File System's ability to read up to 32 Kbytes in one operation, as opposed to the normal maximum of 8 Kbytes using the Berkeley File System. This large limit is practical due to our scatter-gather technique of moving blocks between the buffer cache and disk in clusters, rather than using larger block sizes. The throughput obtained on a similar write test is somewhat lower due to the CPU requirements of allocating blocks, measuring at approximately 380 Kbytes per second on the Quantum disks. It should be noted that these measurements are like EPA mileage estimates. They are obtained under ideal conditions using an empty */tmp* partition to create the file used for the test. Therefore, they represent best case indicators of performance. An additional read throughput data point was obtained from [McKusick84] for the 4.2 BSD File System running on a VAX 11/750 with a MASSBUS controller and Ampex Capricorn 330 Mbyte disk. The following table summarizes the read performance measurements obtained:

Disk Type	Kbytes/sec
Q280	450
Wren III	520
Fujitsu	550
VAX 750	280
McKusick	466

Summary

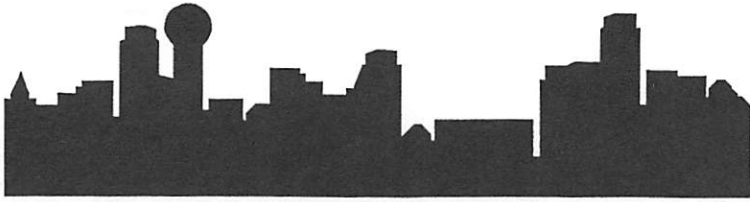
We have shown that the standard System V File System suffers from a severe performance problem when files are accessed sequentially, and presented the details and rationalization for our solution to this problem. Rather than increase the block size, as has been done in other solutions, we chose to transfer multiple blocks between disk and memory. This choice allows us to move up to 32 blocks in a single operation, which is beyond the limit for a reasonable block size in the file system. This advantage allows us to give best-case performance using a SCSI controller that is better than that obtained in much more

expensive computers using SMD technology. The use of a bit map free list allows this feature to be useful by enabling the allocation of adjacent clusters of blocks to files.

Bit map free lists are not without their own problems, however. Allocation of the last few blocks in a file system can become painfully slow, due to the need to scan much of the bit map to find them. This problem can be solved by remembering some of the free block information obtained during scans through the bit map. (The Berkeley file system deals with this problem by reserving a small percentage of the blocks in the file system as not allocatable.) Related to this effect is fragmentation of clusters, where there are no full clusters left for allocation even though there are lots of blocks free. The solution to this problem requires further investigation to find a simple strategy which improves the worst case, which is the normal System V behavior.

References

- [Deitel83] Deitel, H. M., **An Introduction to Operating Systems**, Addison Wesley, 1983, pp. 490-493.
- [Feder84] Feder, J. "The Evolution of UNIX System Performance", Bell Laboratories Technical Journal, Vol. 63, No. 8, October 1984, pp. 1791-1814.
- [Goble81] Goble, G. H. and Marsh, M. H., "A Dual Processor VAX 11/780", Purdue University Technical Report, TR-EE 81-31, September 1981.
- [McKusick84] McKusick, M. K., Joy, W. N., Leffler, S. J. and Fabry, R. S., "A Fast File System for UNIX", ACM Transactions on Computer Systems, Vol. 2, No. 3, Aug. 1984, pp. 181-197.
- [Peacock87] Peacock, J. K., "Application dictates your choice of a multiprocessor model", EDN, Vol. 32, No. 13, June 25, 1987.
- [Powell79] Powell, M., "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov. 1977, pp 33-42.
- [Thompson78] Thompson, K. "UNIX Implementation", Bell System Technical Journal, Vol. 57, No. 6, July-August 1978.



UNIX I/O In a Multiprocessor System

A.J. van de Goor
Delft University of Technology
Department of Electrical Engineering
Mekelweg 4
P.O. Box 5031
2600 GA Delft
The Netherlands
vdgoor@dutesta.UUCP

A. Moolenaar
Oce Nederland B. V.
St. Urbanusweg 126
P.O. Box 101
5900 MA Venlo
The Netherlands
mool@oce.nl.UUCP

ABSTRACT

This article deals with the transfer of some of the I/O functions of the UNIX operating system to I/O processors. The I/O part of the UNIX kernel is examined and a transfer level is chosen in such a way that the number of modifications is minimal. A method (based on horizontal and vertical data sharing) is proposed to determine the transfer level. To facilitate the execution of the transferred functions on the I/O processors, a special kernel with UNIX-like processes is used. A communication mechanism with messages has been designed for communication between the transferred and the non-transferred functions. Although applied to UNIX, the methodology as presented in this paper can be applied anywhere where a large, monolithic program has to be broken down into semi-independent pieces.

Keywords: UNIX I/O, operating system, multiprocessor system, data sharing, intelligent I/O processor.

Introduction

During the last few years the price of microprocessor components has dropped rapidly, while the demand for computer performance has continued to grow. A logical outcome of these two facts is the combination of several microprocessors into one system, to obtain more performance at low cost.

At the Delft University of Technology a project was started in 1984 to build a multiprocessor computer which executes the UNIX operating system. Both the hardware and the software design were part of this project.

To be able to run UNIX in a multiprocessor environment the processing load has to be distributed over several processors. This was achieved by making two types of adaptations:

1. In single processor UNIX, only one process is running at a time. UNIX was adapted to be able to run several processes in parallel.
2. In single processor UNIX, one processor controls

all the I/O. The low level I/O functions of UNIX were transferred to I/O processors (IOPs) to relieve the other processors from handling I/O interrupts and buffering.

This article mainly deals with the second type of adaptation. The first type is only touched upon here, as it is worked out in more detail in [Ja86].

One of the objectives was to adapt UNIX to the multiprocessor environment while making as few modifications as possible. This was necessary because only a limited amount of time (ten months) was available.

In this article a global view of the I/O transfer is given. More details, including an enumeration of all modifications, can be found in the master's thesis "Transferring UNIX I/O to I/O Processors" [Mo85], on which this article is based.

Two types of processors are used in the system: Central processors and I/O processors. One of the central processors is the master. It delegates UNIX tasks

to the other central processors (slaves) and delegates I/O tasks to the I/O processors. On the I/O processors the tasks are executed by a small, UNIX-like, kernel. The communication between master and I/O processors is carried out by a remote calling mechanism. The decision of which tasks had to be transferred was made using a method based on horizontal and vertical data sharing. Although the UNIX kernel is very complex, the method helped in keeping the number of modifications to a minimum.

Section 2 discusses the hardware and software structure of the Delft multiprocessor system. Section 3 introduces the basic ideas behind splitting large modules into smaller ones. The notions of horizontal and vertical data sharing are introduced here. Section 4 describes the I/O structure of UNIX and applies the horizontal and vertical data sharing concepts, resulting in the appropriate transfer level.

Delft multiprocessor system

This section deals with the hardware and software structure of the Delft multiprocessor system.

Hardware overview

The hardware of the multiprocessor system is based on the VMEbus and microprocessor components of the Motorola 68000 series. Three types of boards have been designed:

1. Processor boards with a 68020 microprocessor, a matching Memory Management Unit (MMU) and a software transparent cache.
2. Memory boards with error detection (parity) and support for the software transparent cache.
3. Intelligent I/O boards with a 68010 microprocessor, some memory and device controllers.

A very sophisticated cache mechanism was designed for the system [Go85]. The cache significantly reduces the bus load and is fully transparent to the software. It allows the 68020 to run at maximum speed without using a dedicated bus between CPU and memory.

Software overview

UNIX System V was used during the project, so all UNIX details come from that version. Nevertheless, most of the remarks made here also apply to other UNIX versions.

A UNIX process can be in one of two modes: in *user mode* when the process executes a program and in *kernel mode* when the process executes the UNIX kernel. A process can go from user mode to kernel mode by executing a service request (system call). When the kernel has finished the system call the process returns to user mode.

The simplest method of getting a multiprocessor system is by having only one processor execute processes in kernel mode while all processors are allowed to execute processes in user mode. The processor which executes processes in both user and kernel

mode is called the *master* and the others *slaves*. Consequently, a process can be executed by any processor when it is in user mode, but it must be executed by the master when it is in kernel mode. This master/slave system, designed and implemented by J.K. Annot and M.D. Janssens [An85], is illustrated in figure 1.



Figure 1: Simple multiprocessor configuration

This system is simple because there is only one processor, the master, which executes the kernel. It resembles the single processor situation, so only minor modifications to the kernel have to be made. The master also controls the I/O devices because this must be done by the kernel and the master is the only processor which executes the kernel.

While the number of slaves can be increased, there can only be one master in this configuration. As the reader may quickly conclude, this means that the master will become the bottleneck in this system. Adding many slaves will not do much to improve the overall performance of the system. To gain performance one must relieve the master of some tasks. This can be accomplished in two ways:

1. Let the slaves also execute the kernel. This involves modifying the UNIX kernel in such a way that it can be executed on multiple processors. This is discussed in chapter 9 ("From simple to complex multiprocessor UNIX") of [An85]. This results in a multiprocessor system in which the master and the slaves are equal, except that the master controls the I/O devices.
2. Transfer I/O tasks of the master to I/O processors. This relieves the master from controlling the I/O devices.

The system which results from transferring some

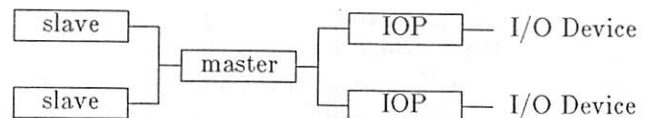


Figure 2: Multiprocessor with I/O processors.

of the I/O tasks to IOPs is illustrated in figure 2. By combining the two methods of improving the simple multiprocessor system, we can omit the master. A system with UNIX processors (UP¹) and IOPs is

¹A UP can execute UNIX processes in both user and kernel mode. So it performs the same tasks as the master in a master/slave system. But because several UPs can work in parallel, this processor can no longer be designated as a master.

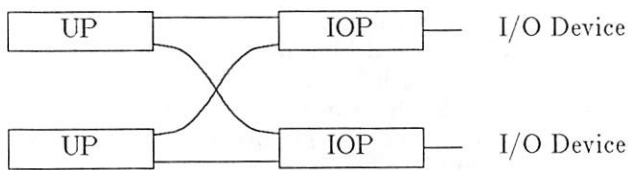


Figure 3: Multiprocessor with UPs and IOPs.

obtained. This is illustrated in figure 3. The performance of this system is not limited by a bottleneck.

In the remainder of this paper the multiprocessor configuration with master, slaves and IOPs of figure 2 will be used. As far as the I/O transfer is concerned, there is not much difference compared to the system with UPs and IOPs of figure 3, because most of the differences are hidden by the communication mechanism (which will be discussed after the section on the IOP kernel).

IOP kernel

To allow for the transferred functions to be executed, a small UNIX-like kernel is used on the IOPs. It is UNIX-like in that the synchronization mechanism is exactly the same as that used in the original UNIX kernel. The processes of this IOP kernel are always in kernel mode and can execute the transferred functions just as if they were not transferred. Because the IOP kernel resembles the UNIX kernel, most functions don't have to be modified when they are transferred.

An IOP process is created when a request enters

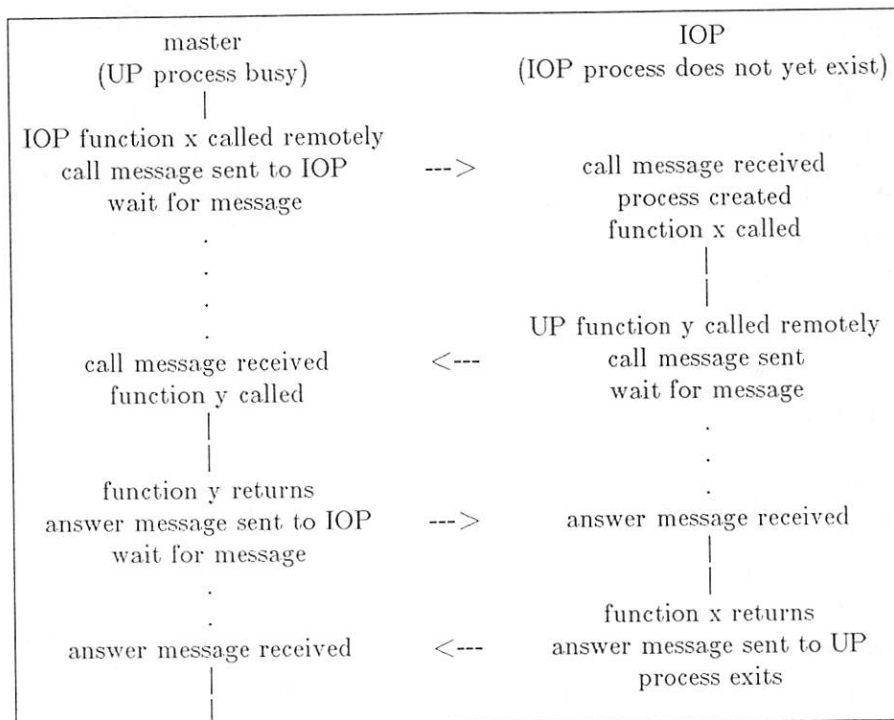
from the master to execute a transferred function. The IOP process executes that function and sends any results back to the requester. The IOP process terminates as soon as the request has been handled.

Communication between master and IOPs

When functions are transferred to IOPs, the non-transferred (UP) functions must be able to call the transferred (IOP) functions. It must also be possible for IOP functions to call some UP functions². This calling from one processor to another processor is done by using a remote calling mechanism.

The order in which the remote calls are executed is fixed. A process executing on the master takes the initiative and calls an IOP function. This function, executed by an IOP process, may then call UP functions. This is illustrated in figure 4. As can be seen in this figure, messages are used by the remote calling mechanism. Each remote function call starts with sending a call message and ends with receiving a corresponding answer message.

²This is necessary for some transferred functions which use non transferred functions. These functions are not transferred because they access the address space of user programs, which cannot be done by an IOP without making a lot of modifications.



"|" = process busy; "." = process sleeping

Figure 4: Remote calling sequence.

Module splitting Criteria

The problems of splitting larger modules into smaller ones are discussed in this section, starting with code-modules followed by data-modules. The concepts of horizontal and vertical data sharing are introduced.

Separation of code

The UNIX kernel consists of a large number of functions. If one wants to transfer a function to an IOP, it must be called from the master by means of the remote calling mechanism. This transferred function may call other functions. Using the remote calling mechanism to call a function on the master from an IOP is undesirable, because this introduces (possibly unnecessary) overhead. Consequently, all the functions that are called directly or indirectly by the transferred function should preferably be transferred too. Unfortunately, there are some functions which cannot be transferred without making a lot of modifications³. If one of these functions is called by a transferred function, a remote call should be used to avoid the modifications, at the cost of some overhead. So we can conclude that if a function is transferred, as many as possible of the functions which are called by that function (directly or indirectly) must be transferred too.

Separation of data

Global data is used by several functions. If some data is used by both a transferred function and a non-transferred function, then there is data sharing between processors. This also occurs if a function on one IOP uses the same data (not necessarily global) as a function on another IOP.

In each function in the I/O part of the UNIX kernel three types of data are used (see figure 5):

1. private data, not shared with any other function
2. data shared with other I/O functions
3. data shared with non-I/O functions

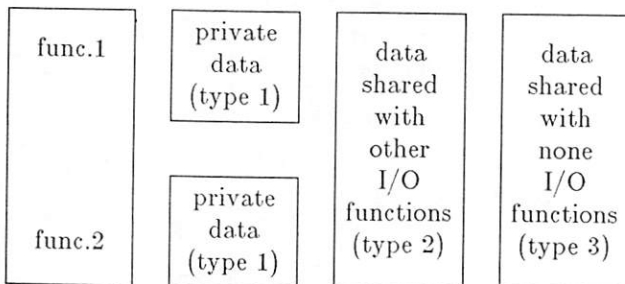


Figure 5: Transferred functions and shared data.

In systems with multiple processors there is always the problem of data which is shared between processors. In many situations this can result in race

³The modifications are mainly caused by the fact that these functions access the address space of a user program, which is very difficult for an IOP.

conditions, which must be avoided by using mutual exclusion to protect the data. When the mutual exclusion mechanism is used by a program, some undesirable side effects, such as starvation and deadlocks, must be avoided. This can be very difficult. Mutual exclusion also decreases performance. Therefore, when a program is adapted from a single processor version to a multiprocessor version, mutual exclusion should be avoided where possible. This can be achieved by avoiding data sharing between processors.

There are two types of data sharing involved in transferring I/O (see figure 6):

1. When some I/O functions are transferred from the master to an IOP, the data types 2 and 3 mentioned above can possibly be accessed by both the master and the IOP. This will be called vertical data sharing.
2. When some I/O functions are transferred to one IOP and some to another IOP, all three data types can potentially be shared between these IOPs. This will be called horizontal data sharing. Horizontal data sharing does not occur for all the data that are used on an IOP, because a large amount of data is used for only one (type of) device. So this data will be used by only one of the IOPs, the IOP which controls that specific device. Other data can be forced to be associated with the devices of one IOP to avoid data sharing. This is especially true for buffers, which will be used here as an example.

In single processor UNIX, buffers are grouped in pools. When a buffer is required, it is taken from the pool and when it is no longer needed, it is returned to the pool. In the multiprocessor system one pool of buffers could be used, but then this pool would be shared by several IOPs, meaning it is necessary to protect it via mutual exclusion. This problem can be eliminated by providing each IOP with its own pool of

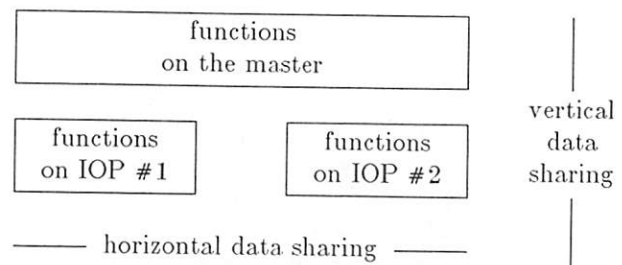


Figure 6: Two types of data sharing.

buffers. Fortunately, this can be realized without changing the buffer request and release functions. A disadvantage, however, is that buffers cannot be shared dynamically between IOPs, but this is compensated by the fact that no time is lost for mutual exclusion and the existing UNIX functions don't have to be modified.

Vertical data sharing cannot be avoided, but can be minimized by properly choosing the transfer level.

If a function shares data with another I/O function (data type 2), there are three possibilities:

1. Both functions are transferred. Then the data is accessed by IOPs only and there is only horizontal data sharing, which has been discussed previously. When the data is associated with a device, horizontal data sharing can be avoided, because this data will only be used by the IOP which controls the associated device.
2. One of the two functions is transferred. This results in vertical data sharing, which creates the need for mutual exclusion and implies making modifications to all the UNIX functions accessing the shared data, unless some sort of mutual exclusion is already present (this holds for block I/O, as will be explained in section 4).
3. Neither function is transferred. Then there is no data sharing at all. This option can be considered if the previous two possibilities don't fit.

The data shared between the I/O functions and the rest of the kernel (data type 3) consists mainly of two groups of data: the user structure and the sysinfo structure. A further examination of this data lies beyond the scope of this article. For the reader it suffices to know that they have no influence on the choice of the transfer level, because they are used in almost every I/O function.

UNIX I/O transfer

This section starts with an overview of the I/O subsystem of UNIX, after which the transfer level of the two main I/O parts (Block I/O and Character I/O) are discussed.

UNIX I/O layers

There are many ways to group the I/O functions of the UNIX kernel. The one presented employs uses a hierarchical model, much like the one used by Pepinck [Pe84]. In this model the I/O is divided into four layers. A function in one layer *uses* functions in the same layer and functions of lower layers. The *uses* relation is defined by the following rule⁴:

Function A *uses* function B if A invokes B and depends upon the results of that invocation.

As an example, suppose there are three layers. The functions in the highest layer may *use* all three layers; the functions in the middle layer may *use* the lower two layers, but will not *use* functions in the highest layer.

In general, the *uses* relation merely means that one function invokes another function. But on some occasions this can yield problems in building a hierarchical model. Suppose function A invokes function B, which invokes function C, which invokes function A again. It is not possible to determine a hierarchy from these invocations. But if one knows that

⁴This definition is based on the original definition of the *uses* relation by Parnas [Pa78].

function C does not depend on the results of function A, we can say that function A lies above function C in the hierarchy.

The separation between I/O layers is not always as clear as it should be. This is due to the fact that the UNIX kernel was not written in a hierarchical way. However, when considering I/O transfer, the following four layers will suffice, see figure 7 (for more details [Pe84] page 111):

1. the system call layer
2. the i-node⁵ layer
3. the buffer cache layer
4. the device driver layer The system call and i-node layers will be called the *higher* layers, by virtue of the fact that these two layers implement the UNIX file system. The *lower* layers, buffer cache and device driver, merely transport data and do not project a structure on this data.

In the *lower* two layers a distinction is made between block I/O and character I/O. These two parts are kept completely separate. Block I/O is done with blocks of data with a fixed length (e.g. for disk transfers) and character I/O with sequences of bytes

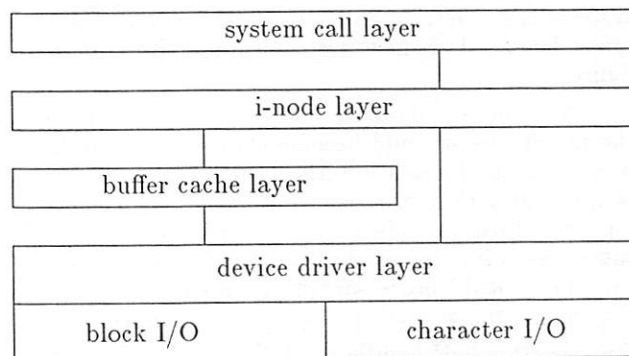


Figure 7: UNIX I/O layers.

of any length (e.g. for communication with terminals). The buffer cache is used for block I/O only. Buffering for character I/O is carried out at the device driver layer.

The global definitions for the I/O layers are listed below:

system call The functions in this layer translate the I/O service requests (system calls) to operations on the actual files in the file system. To allow for this, a data structure is maintained for each process, containing information about all files opened by that process.

i-node The functions in this layer execute the operations on files (which in fact are operations on i-nodes), which result in operations (I/O requests) on devices. It is figured out which device is

⁵An i-node is the internal representation of a file. It is a data structure that contains all attributes of a file (length, creation date, ownership, pointers to the contents of the file, etc.), except for the name.

associated with the file. Data controlled in this layer is the i-node cache⁶.

buffer cache In this layer the buffer cache is controlled. The buffer cache is a software cache mechanism which is used to speed up the access to block I/O devices. It contains the most recently used data of the block I/O devices.

device driver This layer contains all the device drivers. A device driver forms the interface between the device independent part of the kernel and a device. There is one device driver for each type of device.

High transfer level

To relieve the master, as many I/O functions as possible should be transferred to IOPs. But the device on which the I/O will be performed is not known until in the i-node layer (the system call layer doesn't know about devices, it merely uses files). So when the i-node layer is transferred to IOPs, the master does not know which IOP should handle the I/O. Two methods to solve this problem will be considered.

The *first* method is to add a file processor. This processor executes the i-node layer and possibly the system call layer. It calls the IOPs to execute the lower layers. This new architecture is illustrated in figure 8.

A significant disadvantage of this system is that the file processor could become another bottleneck in large systems. In addition, the system gets more complicated and the path from a slave to an IOP gets longer, which introduces extra delays. Because of these disadvantages and because simplicity was one of the goals, the file processor was not used.

The *second* method is to choose an IOP at random, let this IOP handle the I/O down to the i-node layer and then, depending on whether it controls the device or not, continue with the I/O or hand it over to the IOP that does control the device. There are some

⁶The i-node cache is a software mechanism that speeds up the access to the file system by storing the i-nodes of all open files (i.e. files which are opened but not closed yet).

disadvantages to this method:

1. The i-node layer will be executed by all IOPs. Therefore, the data used in this layer, the (software) i-node cache, will be accessed by all IOPs and must be placed in global memory. And because the IOPs, unlike UPs, do not have a (hardware) cache, this will increase the bus load.
2. The I/O will first be handled by one IOP and then possibly handed over to another IOP, which entails extra delays.
3. User address space is accessed in the i-node layer. It is quite difficult to access user space from IOPs. It is certain to cause complexity and may even introduce extra delays.

In light of these reasons the i-node layer was not transferred. The system call layer will not be transferred either, because it is highly illogical and unpractical to transfer the system call layer when the i-node layer is not transferred. The question of whether the lower layers should be transferred or not will be handled for block and character I/O separately in the next two sections.

Block I/O transfer level

There are two I/O layers involved in block I/O: the buffer cache layer and the device driver layer. The device driver layer will certainly be transferred, as otherwise there would be no transfer at all. In order to decide whether the buffer cache layer will be transferred, the horizontal and vertical data sharing must be considered. In figure 9 the main data structures for block I/O are depicted.

Vertical data sharing

As can be seen in figure 9, the buffer cache is accessed in both the i-node layer and the device driver layer. This implies that the vertical data sharing of the buffer cache cannot be avoided, because the device driver layer will be transferred and the i-node layer will not.

Because of this vertical data sharing, the buffer cache should be protected via mutual exclusion. Fortunately, there is already some form of mutual exclusion for the buffer cache. The functions in the i-node

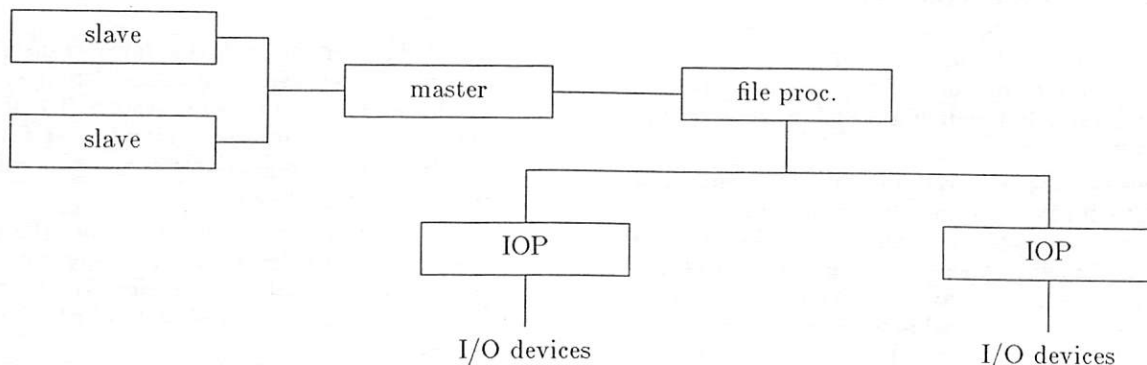


Figure 8: Multiprocessor configuration with file processor.

layer must first request a buffer before they can use it, and they will release the buffer afterwards. This

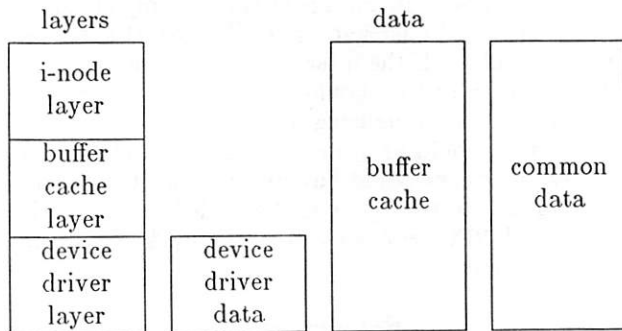


Figure 9: Block I/O transfer levels and data.

mechanism can be utilized for mutual exclusion of the buffer cache. There is yet another mechanism which can be used. Between the buffer cache layer and the device driver layer, buffers are "handed over". The device drivers are invoked to perform I/O on a buffer and when the I/O is complete this is signaled back to the invoking function. This is a very primitive mutual exclusion mechanism.

Because of the two existing mutual exclusion mechanisms, the transfer level could be at the buffer cache layer or at the device driver layer. However, the mutual exclusion mechanism between the i-node layer and the buffer cache layer is more advanced. When vertical data sharing is considered, the buffer cache level is somewhat simpler to implement than the device driver level.

Horizontal data sharing

Another problem occurring when transferring the buffer cache layer is horizontal data sharing. The device driver data should not cause problems, because this data is device dependent, meaning that each IOP can have its own device driver data. The buffer cache could be a source of problems. The buffer cache is a pool of buffers, which are not associated with a specific device. As mentioned in the section on data sharing, this pool could be split into several pools, one for each IOP.

When one buffer cache is used for the whole system, buffers are not statically associated with any device. When one buffer cache is used for each IOP, buffers will always be associated with a group of devices. To what extent the performance is affected by this is hard to tell. It depends on the sequence of accesses to the block I/O devices. It is even possible for performance to be improved, because each IOP could have its buffer cache in local memory, thereby decreasing the bus load. Whatever the performance change, it can always be compensated by modifying the size of the buffer cache (at the cost of some memory, which is assumed to be of minor importance). Therefore the performance aspect will not be taken into consideration when deciding at which level block I/O should be transferred.

When only the device driver layer is transferred, there are two possibilities:

1. Using one buffer cache for each IOP. The buffer cache layer will then have to be modified, because not one but several buffer caches exist, one of which must be selected.
2. Using one buffer cache. The buffer cache layer will then not have to be modified to accommodate the multiple buffer caches, but the access from the device drivers to the buffer cache must be modified at some points for mutual exclusion.

Although it is difficult to choose between these two possibilities, it is clear that modifications cannot be avoided.

When the buffer cache layer is transferred there should be one buffer cache for each IOP, because having one common, system wide, buffer cache entails quite a lot of modifications. The number of modifications involved in multiple instances of the buffer cache (one per IOP) is quite small. This stems from the fact that the higher layers don't know where the buffers of the buffer cache are placed and consequently have to ask this from the functions in the buffer cache layer by requesting a buffer.

When the decision about transfer level is based solely on horizontal data sharing, the buffer cache level is the best choice.

Conclusion

Transferring block I/O at the buffer cache level is a good choice, as it serves both vertical and horizontal data sharing the best.

Character I/O transfer level

The device driver layer for character I/O is quite large. This allows for a split up into several other layers. So one could consider to transfer only a part of the device driver layer. First an important decision has to be made: from which processors can the tty structures⁷ be accessed? This decision is important because the tty structures are used by almost all functions in the character I/O layer. There are three possibilities:

1. If only the master is allowed to access the tty structures, only the functions which don't access the tty structures can be transferred, which are only a few.
2. If both the master and the IOPs are allowed to access the tty structures there is vertical data sharing. Therefore mutual exclusion has to be added, causing a lot of modifications.
3. If only the IOPs are allowed to access the tty structures, then the character I/O layer must be transferred completely. This causes horizontal data sharing. But because the tty structures are each related to a specific device, this horizontal data sharing can easily be avoided.

⁷A tty structure contains all information about a terminal: pointers to the input and output buffers, status, associated processes, etc.

Clearly the third possibility is the best. Transferring the tty structures implies that all character I/O code must be transferred to IOPs. At first glance this might seem to invoke a great deal of work (about 2400 lines of source code are involved), but it actually doesn't. The main advantage is that the module, which is formed by the character I/O code, is not split up by the transfer. As a result it does not matter how character I/O works internally. Only the interfaces between the character I/O part and the rest of the kernel have to be examined, of which there are fortunately only a few.

Conclusion

The major conclusion is that it is possible to transfer some of the I/O functions of the UNIX kernel to I/O processors and make only a small number of modifications. To do this, the transfer level has to be chosen in such a way that data sharing between processors is avoided. The lower I/O layers can be transferred easily with only a small number of modifications, because the data which is used in these layers is related to a device, or can be made to be related to a (group of) device(s), so that this data is used by only one IOP. The higher I/O layers cannot be transferred easily, because the data which is used in these layers is not related to a particular device, so this data would be shared between several processors.

The transfer of the character I/O part is easy to accomplish because there is hardly any data sharing between the character I/O part and the rest of the kernel. One could say that the character I/O part is one module, which contains all the design decisions for character I/O. Once again, this illustrates the advantage of the modularization as proposed by Parnas [Pa71].

The transfer, as it is presented here, has only been implemented partially. The IOP kernel was designed and tested. The communication mechanism was designed but not tested. All modifications which have to be applied to the kernel when the block I/O part is transferred have been identified [Mo85], so no problems should arise during implementation. The modifications which are necessary for the transfer of the character I/O part are not listed, but all areas where problems could arise have been examined and solutions to these problems have been given.

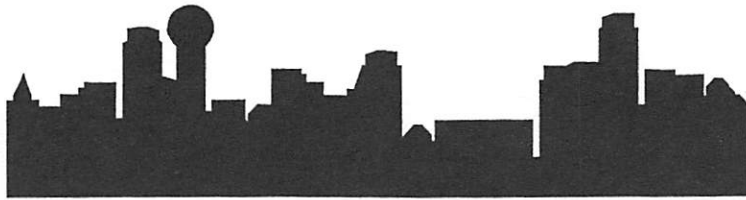
It is difficult to tell how the performance is influenced by the transfer of the I/O functions. At first glance one may be tempted to say that performance actually improves because the master is relieved of some tasks, which frees it to do other things. But the performance of a UNIX system is not just based on processor performance. The throughput of I/O devices, especially disks, is also very important. By having an IOP controlling the I/O device the reaction to a device interrupt will probably be faster. But when the IOP sends an interrupt to the master it is slowed down again. In sum, performance cannot merely be judged by how much is transferred. The

only way to get satisfactory performance figures is to run benchmarks.

An additional advantage of transferring the device drivers to IOPs is that the UNIX kernel gets less device dependent. If the interface between master and IOP is well defined it should be possible to add new IOPs without reconfiguring the UNIX kernel. The device drivers could be present on the IOP in the form of firmware, so one could buy an interface board and just plug it into the system. Nevertheless, the wide variety of UNIX versions will probably keep things from going that far.

References

- [An85] J. K. Annot and M.D. Janssens, *Multiprocessor UNIX*, Master's Thesis, Department of Electrical Engineering, Delft University of Technology, June 1985.
- [Go85] A. J. van de Goor and R. J. Bril, *On Cache Schemes for DUMP1*, to be published.
- [Ja86] M. D. Janssens, J. K. Annot and A. J. van de Goor, *Adapting UNIX for a multiprocessor Environment*, to be published.
- [Mo85] A. Moolenaar, *Transferring UNIX I/O to I/O Processors*, Master's Thesis, Delft University of Technology, Department of Electrical Engineering, July 1985.
- [Pa71] David L. Parnas, *On the criteria to be used in decomposing systems into modules*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, August 1971 (also published in Comm. ACM, vol 15 nr 12, December 1972).
- [Pa78] David L. Parnas, On a "Buzzword": Hierarchical Structure, in: *Programming methodology*, A collection of articles by members of IFIP WG 2.3, edited by David Gries, Springer-Verlag, New York, 1978, pp 335-342; also in: *Proceedings of IFIP Congress 74*, 1974, North-Holland Publ. Co.
- [Pe84] W.R. Peppinck, *Modeling the UNIX kernel*, Master's Thesis, Department of Electrical Engineering, Delft University of Technology, June 1984.



Beyond Threads: Resource Sharing in UNIX

J. M. Barton
J. C. Wagner
Silicon Graphics, Incorporated
2011 Stirlin Road
Mountain View, California 94039
jmb@sgi.com, jwag@sgi.com

ABSTRACT

UNIX provides a programming model for the user which gives an illusion of multiprocessing. On uniprocessors, this illusion works well, providing communication paths, firewalls between processes and a simple programming environment. Unfortunately, the normal UNIX model does not provide the capabilities to take full advantage of modern multiprocessor hardware. This results from a design which uses data queueing and preemption to provide the multiprocessing illusion, which are unnecessary on a true multiprocessor.

The recent proposed addition of *threads* to CMU's MACH kernel shows that there are other effective programming models that may be supported in UNIX as well. This model provides for sharing the virtual address space of a process by breaking the process into several lightweight contexts that may be manipulated more quickly than a normal process. Unfortunately, this model raises questions about support for normal UNIX semantics, as well as thread scheduling performance and kernel overhead.

This paper describes a programming model which goes beyond the simple concept of threads, providing a much broader range of resource sharing while retaining the key parts of the UNIX process model. Instead of *threads*, the concept of a process is retained along with most of its semantics. As usual, the fundamental resource shared is the virtual address space. In addition, open file descriptors, user and group ID's, the current directory, and certain other elements of the process environment may be shared also. This makes for easy construction of useful servers and simple concurrent applications, while still providing high-performance support for more computationally intensive applications.

This implementation, labeled *process share groups*, allows normal process actions to take place easily, such as system calls, page faulting, signaling, pausing, and other actions which are ill-defined within the *threads* model. The paper describes the philosophy which drove the design, as well as details of the implementation, which is based on, and upwardly compatible with, AT&T 5.3 UNIX. Performance of normal UNIX processes is maintained while providing high-performance share group support.

Introduction

The success of the UNIX kernel is owed in part to the effective way in which it hides the resource sharing necessary in a multiprogrammed environment. A UNIX process is a highly independent entity, having its own address space and environment. Communication paths are restricted to low-bandwidth mechanisms, such as pipes, sockets and messages.

When moved to a multiprocessor, this model is overly restrictive. We would like to allow several processors to work on a problem in parallel using high-bandwidth communications (shared memory, for instance). Even though this explicit sharing of data is thought of most often when working with parallel programming, there are other resources that can be

shared usefully. For instance, the file descriptors associated with a process could be shared among several processes.

In order to provide a conceptual framework for resource sharing among concurrent processes, we have introduced an additional level of functionality into the normal UNIX process model. A process may gain access to this additional level of functionality through an interface called *process share groups*. Members of a *share group* can potentially share many resources previously thought of as private to a single process.

This interface was first demonstrated in an AT&T System V.3 kernel, and relies on modified region handling abilities to share a virtual address space. New file opens can be propagated to all

processes, as well as modifications to the environment of a process (such as the *ulimit(2)* values). By extending the semantics of the UNIX process in an upwardly compatible way, a powerful new programming model has been developed.

The Road to Resource Sharing

In order to provide a simple model for multi-programming, the designers of UNIX chose a model of

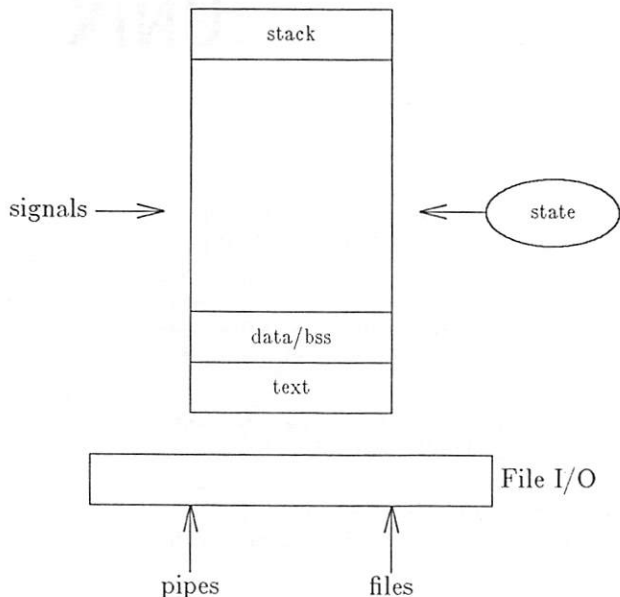


Figure 1: Version 7 Process Environment

independent processes, shared access to a file-system, and limited communications using low-bandwidth paths, such as signals or pipes (Figure 1). Such a decision was appropriate, since the inherent queueing of data implied at many levels of the UNIX interface simplifies multi-programming support. To meet the growing need for high-performance and concurrent programming support, some new mechanisms were introduced. Berkeley UNIX introduced the *socket* interface, while System V introduced a local inter-process communication interface providing shared memory support (Figure 2). No new interfaces for managing files were proposed, nor were changes to improve concurrent programming abilities put forward. Most recently, the Mach[Acce86] kernel introduced an implementation of tasking labeled *threads* [Teva87]. This model allows an address space to be shared among several processes. Each process can execute independently in both kernel and user mode. Although independent execution adds additional cost for a threaded process, such as kernel context (the user area) and a kernel stack for each thread, a useful concurrent programming environment is provided (Figure 3).

As part of the research for share groups, a true threaded process implementation on System V.3 was constructed[Wagn87]. This implementation of

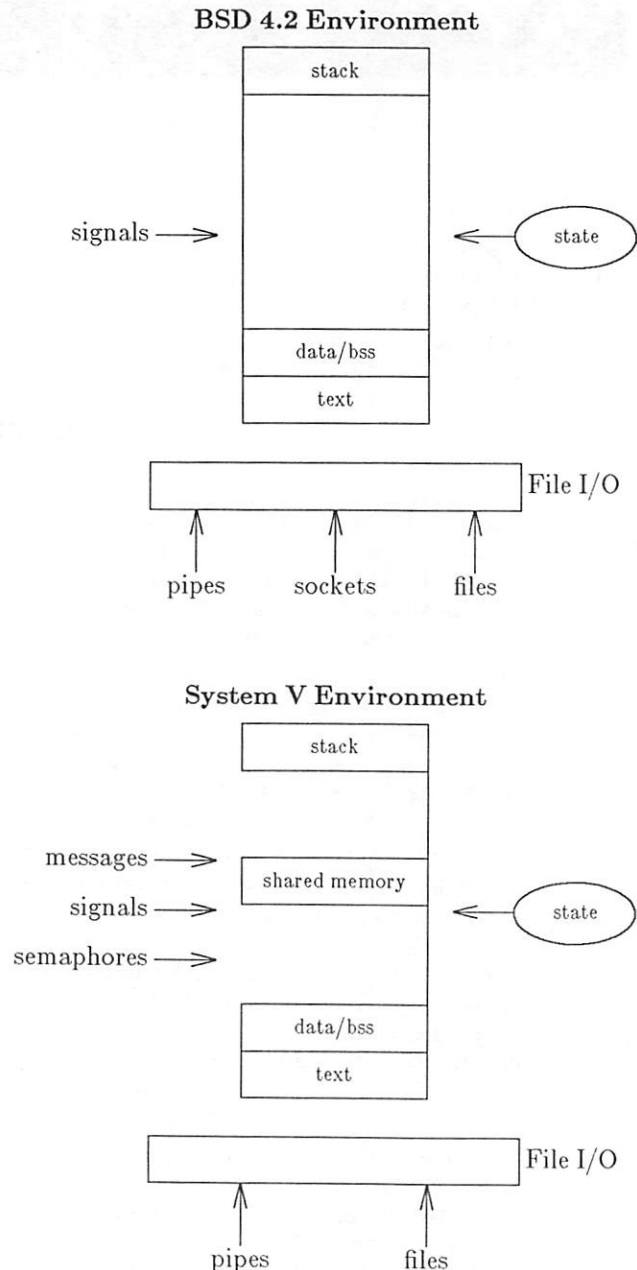


Figure 2: System V and BSD Process Environments

threaded execution went beyond that used in the Mach kernel. A thread was realized as an actual entity within a process, and truly manifested itself only as the minimum machine context which could be scheduled. This version of threads provided a good environment for numerical or other computationally intensive algorithms, but it suffered from limited applications. Finally realizing the limitations of this approach, we looked for a more powerful model.

Parallel Programming

To take maximum advantage of multi-processors, it must be possible to create and execute parallel algorithms, and to get truly parallel execution of application code. Because of the data-sharing

bandwidth necessary for applications to gain performance from parallelism, shared memory is usually the main data path. Sharing memory isn't enough, however. The sharing mechanism and other aspects of the environment must all work together to provide a useful programming model[Bart87].

Shared memory is not useful without a mechanism for synchronizing access to it. For instance, pipes, System V messages or semaphores, sockets or signals can be used to synchronize memory access between processes, but are all much lower bandwidth mechanisms than the memory itself, which can be a liability. With hardware supported locking, synchronization speeds can approach memory access speeds.

Two equally important aspects of the environment are scheduling and context switching. Even if a multi-process application limits itself to the number of processors available, the best performance will occur only if all the processes are actually running in parallel. The kernel needs to take steps to insure that

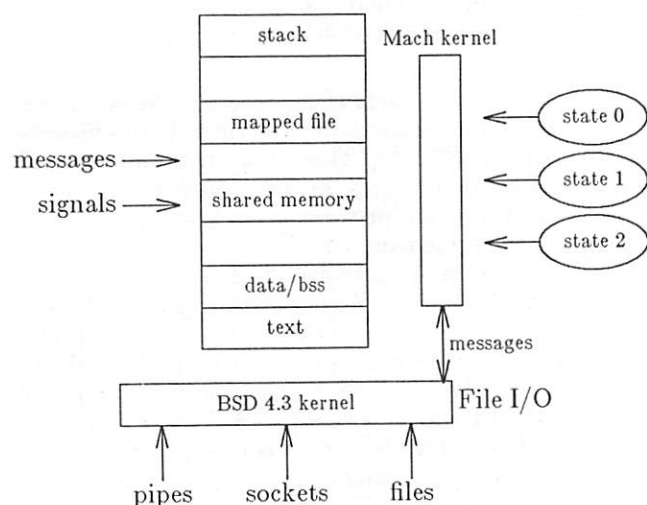


Figure 3: Mach Process Environment

these processes run in parallel whenever possible. Since this isn't always possible in UNIX, fast context switching is also important. For high-performance semaphores, it is necessary that blocking or unblocking a process be extremely quick. Fast context switching is one element of this; the kernel must provide facilities for the actual blocking operations to be executed quickly as well.

Dynamic creation and destruction of processes must be possible, and getting an existing process started on a new task must have little overhead. In a threaded model, creation of a new thread is often an order of magnitude faster than creation of a process, however this seems unimportant. If a process model is used instead (such as Sequent's[Beck87]), then the speed penalties of process creation are eliminated by creating a pool of processes before entering parallel sections of code, each of which then waits to be dispatched quickly as needed. If enough processes are not available, a new one can be dynamically created,

but this problem can be tuned out of the application.

If we add to this model by introducing sharing of other process resources, we approach a useful superset of normal UNIX process semantics. Signals, system calls, traps and other process events happen in an

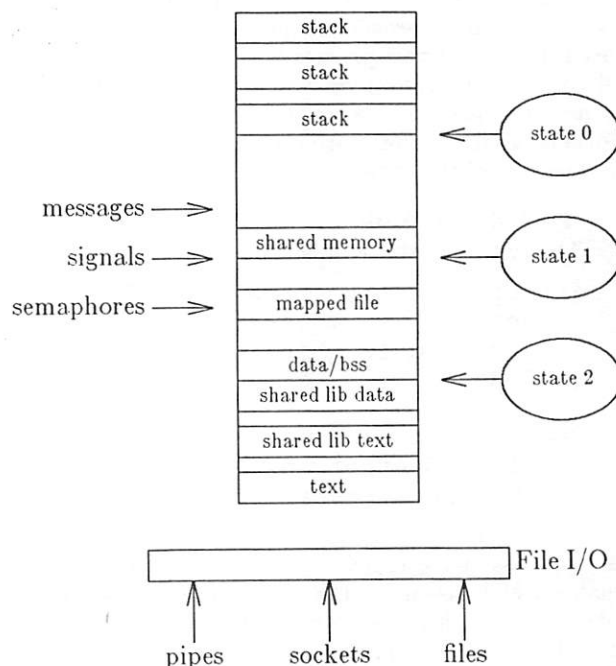


Figure 4: New SGI 4D Series Environment

expected way, since they deal with actual processes. By sharing other resources, such as file descriptors, user ID's, and the like, we can develop a powerful model that gives the performance of threads with greater functionality (Figure 4).

Share Groups

To address the failures and limitations of resource sharing when applied to multiprocessors, we developed the concept of the *share group*. Such a group is a collection of processes which share a common ancestor and have not executed the *exec(2)* system call since being created. The parent-child relationship between processes is maintained, and forms the basis for a hierarchical resource sharing facility, where a parent can indicate what shared resources each child should share.

In general, the main resource shared is the virtual address space associated with the share group, although it need not be. Processes in the share group are free to pass pointers to data, and to use high-performance synchronization and data passing techniques (mainly shared memory and spinlocks).

A small number of other resources may be shared in the initial implementation. Chief among these are file descriptors. When one of the processes in a group opens a file, the others will see the file as immediately available to them. The descriptor number¹ may be passed between processes or simply assumed as part of

the algorithm. Other resources that may be shared are the *ulimit* values, the *umask* value, the current directory and the root directory

System Call Interface

The share group interface is defined through two new system calls, *sproc()* and *prctl()*. The *sproc()* call is used to create a new process within the share group, and controls the resources which will be shared with the new process. The *prctl()* call is used to obtain information about the share group and to control certain features of the group and new processes.

The *sproc* System Call

The syntax of this call is:

```
int sproc(entry, shmash, arg)
void      (*entry)();
unsigned long shmash;
long      arg;
```

returns -
 -1 - OS error in call
 >0 - new process ID

This call is similar to the *fork(2)* system call, in that a new process is created. The *shmash* parameter specifies which resources the child will share with the share group, each particular resource being identified with a bit in the parameter. A new stack is automatically created for the child process, and the new process is entered at the address given by *entry*. This new stack is visible to all other processes in the share group, and will automatically grow. The single argument *arg* is passed to newly created process as the only parameter to the *entry* point, and can be used to pass a pointer to a data block or perhaps an index into a table.

The first use of the *sproc()* call creates a *share group*. Whenever a process in the share group issues the *sproc()* call the new process is made part of the parent's share group. A new process may be created outside the share group through the *fork(2)* system call, and use of the *exec(2)* system call also removes the process from the share group before overlaying the new process image.

All resource sharing is controlled through the *shmash* (**share mask**) parameter. Currently, the following elements can be shared:

```
PR_SADDR - share virtual address
           space
PR_SULIMIT - ulimit values
PR_SUMASK - umask values
PR_SDIR - current/root directory
PR_FDS - open file descriptors
PR_SID - uid/gid
```

¹The descriptor number is an index into the file table for a process, which holds pointers to open file table entries. For example, *standard input* is by convention descriptor number 0, while *standard output* indicates descriptor number 1.

PR_SALL - all of the above and any future resources

When the child is created, the share mask is masked against the share mask used when creating the parent. This means that a process can only cause a child to share those resources that the parent can share as well, thus forming a hierarchy. The original process in a share group is given a mask indicating that all resources are shared.

Whenever a process modifies one of the shared resources, and its share mask indicates that it is sharing the resource, all other processes in the share group which are also sharing the resource will be updated.

When the virtual address(VM) space is shared, a process may modify the VM space and all other processes will see that same change. If new regions are added to the VM space, the kernel will check to insure that no address range overlaps any other. If the virtual address space is not shared, the new process (child) gets a *copy-on-write* image of the share group's (parent's) virtual address space. In this case, the child's stack is not visible in the parent's virtual address space.

Certain small parts of a process's VM space are not shared. The most critical of these is the process data area, or PRDA. This is a small amount of memory (less than a page in size) which records data which must remain private to the process, and is always at the same fixed virtual location in every process. As an example, consider the *errno* variable provided by the C library. Since the data space is shared, if this variable were only in the data space it would be difficult for independent processes to make reliable system calls. Thus, the C library locates a copy of *errno* in the PRDA for a process. The format and use of the PRDA is totally within the scope of the user program, and can be managed in any way appropriate. Libraries (such as the C library) which need some private data may use a portion of the PRDA, leaving a portion for direct use by the programmer.

The *prctl* System Call

The second system call is the *prctl()* call, which allows certain aspects of a share group to be controlled. Its syntax is:

```
int prctl(option [, value [, value2 ]])
unsigned option;
char *value;
char *value2;
```

returns -
 -1 - error in OS call
 <>0 - request result

The following *option* values may be used:

```
PR_MAXPROCS - return limit on
              processes per user
PR_MAXPPROCS - return number of
              processes that the system can
              run in parallel.
```


PR_SETSTACKSIZE - sets the maximum stack size for the current process.

PR_GETSTACKSIZE - retrieves the maximum stack size for the current process.

The PR_SETSTACKSIZE option allows the program to set the maximum stack size which it may have. This value is inherited across *sproc()* and *fork()* system calls, and indirectly controls the layout of the shared VM image.

Implementation

The implementation of share groups centered on four goals:

1. The implementation must work correctly in both multi-processor and uni-processor environments.
2. Synchronization between share group processes must be able to proceed even though one member is not available for execution.
3. The overall structure of the kernel must not be modified.
4. The performance for non-share group processes must not be reduced.

The multiprocessor requirement proved to be the major difficulty in the design, since there are no formal interfaces for synchronization between processes executing in the kernel. This is not usually a problem on uni-processor systems since a process executing in the kernel cannot be preempted. Thus it may examine and modify the state of any other process, and know that the state will not change (unless the

executing process goes to sleep). This isn't true in a multi-processor environment. The process being examined may be running on another processor, sleeping on a semaphore waiting for a resource (it could even be waiting for a resource that the examining process controls), or it may be waiting for a slow operation (i.e. read on a pty, performing the *wait(2)* system call, etc.).

Data Structures

For each share group, there is a single data structure (the shared address block) that is referenced by all members of the group. See the typedef structure on this page for a complete definition. This structure is dynamically allocated the first time that a process invokes the *sproc(2)* system call. A pointer in the *proc* structure points to this, and the *s_plink* field links all processes via a link field in the *proc* structure. To protect the linked list during searching, the lock *s_listlock* is used. A reference count is kept in *s_refcnt*, and the structure is thrown away once the last member exits (Figure 5). The rest of the fields are used to implement the resource sharing between group members and are explained in the following sections.

Virtual Space Sharing

The kernel in which this was implemented is based on System V.3, and therefore uses the *region*[Bach86] model of virtual memory. This model consists of 2 main data structures - *regions*, which describe contiguous virtual spaces (and contain all the page table information etc.) and *pregions*, which are linked per-process and describe the virtual address at which a region is attached to the process, and certain other per-process information about the region of memory. This model is designed to allow for

```
typedef struct shaddr_s {
    /* the following are all to handle preregions */
    preg_t    *s_region;        /* processes' shared preregions */
    lock_t     s_acclock;       /* lock on access to shared block */
    sema_t     s_updwait;       /* wait for update lock */
    short      s_accnt;         /* count of readers */
    ushort     s_waitcnt;       /* count of waiting processes */
    /* generic fields for handling shared processes */
    struct proc s_plink;        /* link to shared processes */
    ushort     s_refcnt;        /* # process in list */
    ushort     s_flag;          /* flags */
    lock_t     s_listlock;      /* protects s_plink */
    /* semaphore for single threading open file updating */
    sema_t     s_fupdsema;      /* wait for opening file */
    struct file **s_ofile;      /* copy of open file descriptors */
    char       *s_pofile;       /* copy of open file flags */
    struct inode *s_cdir;       /* current directory */
    struct inode *s_rdir;       /* root directory */
    /* lock for updating misc things that don't need a semaphore */
    lock_t     s_rupdlock;      /* update lock */
    /* hold values for other sharing options */
    short      s_cmask;         /* mask for file creation */
    daddr_t    s_limit;         /* maximum write address */
    ushort     s_uid;           /* effective user id */
    ushort     s_gid;           /* effective group id */
} shaddr_t;
```

full orthogonality between regions that grow (up or down), and those that can be shared.

The most interesting (and difficult) part of resource sharing to implement is sharing of the VM image, although the job is simplified by using *regions* as the basis for managing the VM. The *sproc* system call shares code with the standard *fork* call, the only difference being the handling of regions. If address space sharing is not indicated when the *sproc*() call is made, then the standard *fork*() operations are performed, generating an image that provides *copy-on-write* access to the VM image. If sharing of the VM image is specified, the private regions of the parent process are marked as *copy-on-write* in the child, while all other regions are shared. A point to remember is that a *fork*() or non-VM sharing *sproc*() call leaves any visible stack or other regions from the share group as *copy-on-write* elements of the new process.

Algorithmically, the private regions for a process are examined when demand paging or loading a new process before examining the shared regions. This provides the *copy-on-write* abilities of a non-VM sharing share group member. It also provides a basis for future enhancements to the manner in which the VM is shared. For instance, it could be possible to share part of the VM image and have *copy-on-write* access to other parts of the image.

Sproc() allocates a new stack segment in a non-overlapping region of the parent's virtual address space. The new process starts on this stack and therefore does not inherit the stack context of the parent (though the child can reference the parent's stack). The child process starts executing at the address given in the *sproc*() call.

Unfortunately, the stock (V.3) region implementation does not support growing or shrinking shared

regions (e.g., the data and BSS portion of a process). The main problem is that although regions have a well defined locking protocol, a basic assumption is made that only the process owning a private region (one that has only a single reference) will ever change it. Using this assumption, various parts of the kernel hold implicit pointers into the region (i.e. pointers to pages) even though they no longer hold a lock on the region. If a process comes in and shrinks such a region, then any implicit pointers are left dangling. Lack of adequate region locking also comes into play when scanning a pregon list attempting to match a virtual address reference with a region of actual memory (during a page fault, for example). This list is not usually protected via a lock since only the calling process can change it. With growable sharable regions, information that is used during the scan could change. The former problem is inherent in uni-processor as well as multi-processor systems, and the later is only a problem for multi-processor systems.

The obvious solution to this is to use a lock on the pregon list. Since all share group processes must obtain this lock each time they page fault, the lock was made a shared read lock - any number of processes can scan the list, but if a process needs to update or change the list or what it points to, it must wait until all others are done scanning. Unfortunately, to guard against the 'implicit' references mentioned above, the lock could not be hidden in the existing region lock routines, but had to be placed around the entire sections of code that reference the virtual address.

The shared read lock consists of a spin lock *s_accclk* which guards the counters: *s_accnt* and *s_waitcnt*. *s_accnt* is the number of processes reading the list (or -1 if someone is updating the list); *s_waitcnt* counts the number of processes waiting for

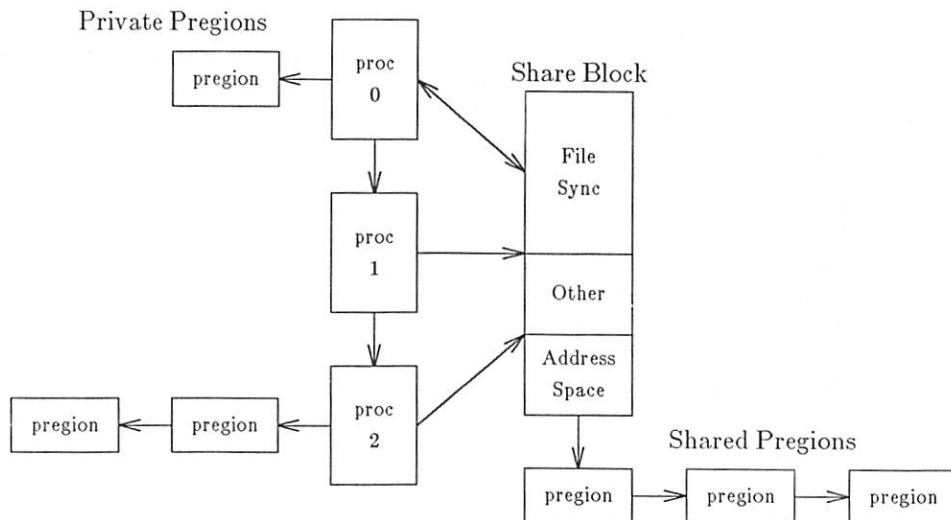


Figure 5: Share Block Data Structure

the shared lock. *s_updwait* is a semaphore on which waiting processes sleep. Since operations that require the update lock are relatively rare (fork, exec, mmap, sbrk, etc) compared to the operations that scan (page fault, pager) the shared lock is almost always available and multiple processes do not collide.

When a process first creates a share group (by calling *sproc(2)*) all of its sharable preions are moved to the list of preions in the shared address block. Some types of regions are not currently shareable. For instance, private text regions may be created for debugging - that way breakpoints may (but are not required to) be set in an individual share group member. The shared preion list is protected via the shared lock in all places that the preion list is accessed. In most cases the only code change was to put calls to the shared lock routines around the large sections of code that reference the preion or region. Since there is only one list, if one process adds a preion (say through a *mmap(2)* call) all other share group members will immediately see that new virtual region.

An interesting problem occurs when a process wishes to delete some section of its virtual space either by removing or shrinking a region. In this case it is important that the actual physical pages not be freed until **all** share group members have agreed to not reference those pages. Also important is that the initiating process not have to wait for each group member to run before completing its system call (some members may not run for a long time). To solve this problem, we use the fact that the TLB (translation lookaside buffer) is managed by software for the target processor (a MIPS R2000[MIPS86]). Thus before shrinking or detaching a region, we synchronously flush the TLBs for ALL processors, while holding the update lock for the share group's preions. Thus if any group members are running, they will immediately trap on a TLB miss exception, come into the kernel and attempt to grab the shared read lock and block. Since the lock is held for update, the process will sleep until the lock is released. The invoking process may then continue to remove the region, free the lock and leave the kernel.

Other Attribute Sharing

Unlike virtual space, other process resources are not visible outside of the kernel, thus it is only important that they be synchronized whenever a group member enters the kernel. As with virtual space synchronization, we don't want to force the calling process to wait until all other group members have synchronized. To implement this, we keep a copy of each resource in the shared address block: current directory inode pointer, open file descriptors, user ids, etc. This also allows immediate access to this data, as most of it is kept in the user area and therefore inaccessible to other than the owning process.

Those resources which have reference counts (file descriptors and inodes) have the count bumped one for the shared address block. This avoids any races

whereby the process that changed the resource exits before all other group members have had a chance to synchronize. Since there always exists a reliable available copy of the data, all that remains is to synchronize the data on each member's entry to the kernel. To make this efficient, multiple bits in the proc structure *p_flag* word are used. When a group member changes a sharable resource, it first checks its *p_shmask* mask (the kernel version of the share mask) to see if it is sharing this particular resource. If so, *s_fupdsema* or *s_fupdlock* is locked to avoid any races between multiple updaters. The resource is then modified (open the file, set user id, etc), a copy is made in the shared address block, each sharing group member's *p_flag* word is updated, and the locks are released. When a shared process enters the system via a system call, the collection of bits in *p_flag* is checked in a single test; if any are set then a routine to handle the synchronization is called. Other events that must be checked on entry to the system were also changed to this scheme, thus lowering the system call overhead for most system calls.

The above scheme works well except if two processes attempt to update a resource at the same time. The lock will stop the second process, but it is important that the second process be synchronized prior to being allowed to update the resource. This is handled by also checking the synchronization bits after acquiring the lock.

Analysis

The resource sharing mechanism described above was implemented and tested on a multi-processor MIPS R2300 testbed. As expected, the time for a *sproc()* system call is slightly less than a regular *fork()*. The overhead for synchronizing the virtual spaces is negligible except when detaching or shrinking regions. In practice this only happens if a process shrinks its data space (fairly rare) or if a process does considerable VM management activity (e.g. mapping or unmapping files). A possible system performance gain could be to only selectively flush the TLBs of the other processors.

Future Directions

Although the set of features included in the first release is adequate for many tasks, there are many other interesting capabilities that could be added.

For example, the ability to selectively share regions when calling *sproc()* could be a useful facility, somewhat along the lines of Mach shared memory, thus allowing some parts of the address space to be accessed *copy-on-write* while others are simply shared. This ability is a simple extension to the current scheme, as it only requires proper management of the private preion list and the shared preion list.

There are also other resources that could be usefully shared. For instance, the scheduling parameters of a process could be shared among the members of

the share group. Since the shared address block is always resident, it provides a convenient handle for making scheduling decisions about the process group as a whole. In a multi-processor example, the programmer could specify that at least two of the processes in the share group must run in parallel, or the group should not be allowed to execute at all. The priority of the whole group could be raised or lowered, or a whole process group could be conveniently blocked or unblocked.

Currently, calling *exec(2)* breaks the association with the share group. By modifying the concept of pregon sharing to handle a unique address space, it could be possible to have a group of unrelated programs managed as a whole for file sharing or scheduling purposes.

Conclusion

This paper has presented a new and unique interface for the System V kernel, *share groups*, that allows an extremely high level of resource sharing between UNIX processes. The programmer has control of what is being shared. Normal UNIX semantics are maintained for all processes within a share group, thus maintaining compatibility and expected behavior.

This interface has been implemented within a production kernel, and meets the promise of its design. Processes not associated with a share group experience no penalty for the inclusion of share group support, while processes within a share group may take advantage of a common address space and automatic sharing of certain fundamental resources.

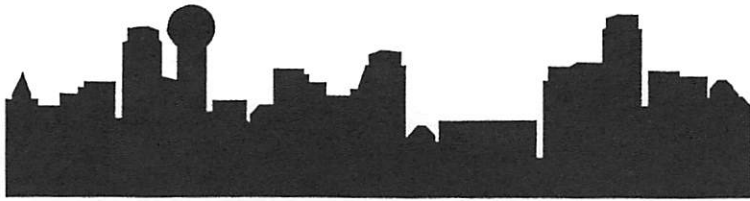
This interface also allows for a large degree of future extensions, and shows how an additional layer of process management may be added to the UNIX kernel without penalizing standard programs.

References

- Accetta, Mike, et al., "Mach: A New Kernel Foundation For UNIX Development", in *USENIX Association Proceedings, Summer, 1986*.
- Tevanian, Avadis, et al., "Mach Threads and the UNIX Kernel: The Battle for Control", in *USENIX Conference Proceedings, Summer 1987*.
- Wagner, J. C., and Barton, J. M., "Threads in System V: Letting UNIX Parallel Process", a work-in-progress paper presented in *login.*, USENIX Association, Vol. 12, No. 5, September/October 1987.
- Barton, J. M., "Multiprocessors: Can UNIX Take the Plunge?", in *UNIX Review*, October, 1987.
- Beck, Bob, and Olien, Dave, "A Parallel Programming Process Model", in *USENIX Conference Proceedings, Winter, 1987*.

Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice Hall, New Jersey, 1986.

MIPS R2000 Processor Architecture, Mips Computer Systems, Inc., Mountain View, CA., 1986.



Watchdogs: Extending the UNIX File System

Brian N. Bershad
C. Brian Pinkerton
University of Washington
Department of Computer Science FR-35
Seattle, Washington 98195
(206) 545-2675
brian@june.cs.washington.edu
bp@june.cs.washington.edu

ABSTRACT

The traditional UNIX file system provides operations whose semantics are fixed at file system implementation time. *Watchdogs* are user-level processes that can extend the file system to achieve user-defined file system semantics on a per-file basis. Watchdogs provide only those functions that need special handling. Other operations proceed through the normal file system, unimpeded by the existence of watchdogs. We describe a prototype implementation of watchdogs that has been used to build several useful applications. Although only a prototype, the system has acceptable performance.

Introduction¹

The traditional UNIX file system serves as a repository for passive data objects – files. It provides facilities for naming, protecting, storing and accessing these files. The file system defines the semantics in each of these areas, constraining users to the design decisions of the file system's implementors.

Watchdogs are extensions to the 4.3BSD UNIX file system that allow users to define and implement their own semantics for files. A watchdog is a user-level program associated with a file or directory. This program can provide alternative implementations of the file system's naming, protection, storage and access functions for that file or directory. The distinguishing characteristics of watchdogs include

- the ability for unprivileged user-level programs to serve as watchdogs,
- added expense only for those functions that are redefined,
- complete transparency to programs accessing associated files, and
- interprocess communication cost commensurate with that of the other major IPC mechanism – pipes.

Watchdogs can provide users with the types of features generally made impossible by the requirements that a file system be at once fast, simple and

general. For example, a compaction watchdog can transparently compact and uncompact a file as it is being written and read. Users can specify their own file protection policies by creating a watchdog that redefines the *open* system call. Intelligent mailboxes, where actions are taken automatically upon the receipt of new mail, are also possible using watchdogs. Watchdogs make file versioning possible without burdening either the operating system (as in VMS VMS or the user (as in SCCS scs with the responsibility of maintaining past versions. Watchdogs also facilitate the creation of special-purpose pseudo file systems, where the files that comprise the file system need not exist within any single UNIX file system. To this end, watchdogs have been used as part of the Heterogeneous Computer Systems project at the University of Washington Interconnecting Heterogeneous Computer Systems, black, levy, zahorjan to allow UNIX machines to import files from heterogeneous file servers.

Watchdog Operation

The central object in the extended filesystem is the *guarded* file. A guarded file is an existing file or directory with which a watchdog has been associated. This watchdog is notified every time the file is opened and interacts with the kernel to provide extended semantics for the file.

A file consists of some data and some operations allowing processes to access that data. In the absence of watchdogs the implementation of any one file's operations are common to all files in the system. Watchdogs give each file the chance to provide its own implementation of procedures that implement the

¹This work took place while the authors were supported by NSF Grants No. DCR-8420945 and CCR-8611390. Bershad was also supported by a USENIX Association fellowship. Computing equipment was provided by Digital Equipment Corporation's External Research Program.

standard file system interface (open, close, read, write, etc.). Figure 1 diagrams the relationship between processes, the file system and watchdogs.

Watchdogs can interact with user-level processes only through the kernel because their involvement must be transparent to the user program. The strength of this transparency is demonstrated in two ways: user programs need not be recompiled to take advantage of watchdogs, and user programs cannot circumvent the watchdog system to access files directly.

When a file guarded by a watchdog is opened, the opener is suspended and the watchdog is notified of the open by a message from the kernel. The message contains the arguments to the open call and the identity of the opener. With an acknowledgment to the kernel, the watchdog either denies or permits the requested access. Should access be granted, the watchdog informs the kernel of any other operations *on the opened file* that should be guarded by the watchdog. A watchdog may guard different instances of an open file in different ways. This allows a single file to have multiple views.

A request for a guarded operation on the opened file is relayed to the watchdog for processing. Upon being notified of the request, the watchdog must do one of the following:

- i) perform the operation, providing or consuming any data normally associated with the operation. For example, a watchdog would satisfy a read request by returning a buffer of data to the reading process via the kernel. Whether the file originally opened actually sources or sinks this data depends on the implementation of the watchdog. To avoid loops, watchdogs are permitted direct access to the files they guard.

- ii) deny the operation, replying to the kernel with the UNIX error code that should be relayed to the process requesting the operation.

- iii) defer the operation, when possible, back to the kernel. In this case, the watchdog simply acknowledges the operation, but relies on the kernel to actually perform it. Deferment is appropriate when only the occurrence, but not the actual function, of an operation deserves attention (such as in accounting applications).

In all three cases, the process requesting the operation is unaware that its request is being evaluated by another process. Since only guarded operations are relayed to the watchdog, others proceed at their full speed, unimpeded by the existence of the watchdog.

Directory Watchdogs

Watchdogs can be associated with directories as well as with files. When a process opens a file, the process's access rights are checked for each directory in the file's pathname. If any of these directories is guarded by a watchdog, the watchdog is asked to validate the access attempt. This may require that several watchdogs be consulted during the resolution of a single pathname. If the parent directory of an opened file is guarded, and the file does not have an explicit watchdog, the parent directory's watchdog is used to negotiate access to the opened file. This arrangement allows one watchdog to collectively manage all files in a single directory.

Watchdogs permit a file's contents to be illusory. A process sees whatever the watchdog chooses to let that process see. Directories are no exception. A process can open a physically non-existent file in a physically non-existent directory as long as the last

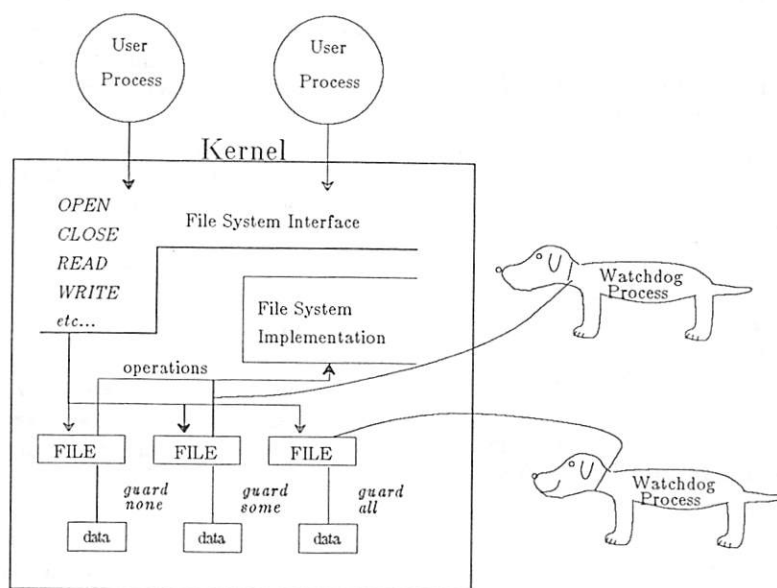


Figure 1 - Processes, Watchdogs and the File System

resolvable component of that file's name has an associated watchdog capable of handling the deception. Note that in this case all operations must be guarded, and the watchdog may not defer any activity back to the kernel.

To demonstrate the behavior of the different styles of watchdog attachment, consider the directory tree shown in Figure 2. A user trying to open `/X/file1` would first be authenticated against *bowser*. If access to the directory is granted, *fido* is notified of the open attempt and can then inform the kernel if final access is approved. In the case of `/X/file2`, *bowser* is responsible for both granting access to the file and assuming any of the file system operations that it chooses. This is because `file2`, though unguarded, lives in a guarded directory. A user trying to open `/X/Z/file3` will only be checked for access when passing through `/X`. The file itself is unguarded. Finally, attempts to access `/Y/file4` or `/Y/file5` would be handled by the watchdog *spot*.

Related Work

Many other systems have concentrated on moving a piece of the file system up to the user-level in order to obtain some new functionality. The principal difference between watchdogs and these efforts is that watchdogs provide a kernel framework for such extension while the others build a veneer on top of existing kernel facilities. For example, IBIS IBIS and the Newcastle Connection newcastle connection implement remote file systems at user-level by modifying the standard subroutine libraries. The Apollo DOMAIN system apollo, extended naming allows users to define typed objects (files) and operations (procedures) to manipulate these objects. The system relies on dynamic loading to bind an executing program to the appropriate implementation of an object's procedures.

The advantage of these other systems is that the interface and implementation of the operations are coresident in a single process so there is no context switch or IPC overhead. Unfortunately, users are still able to circumvent the extended interface, accessing

files directly using the basic file system calls, or they may be prevented from accessing their files unless explicitly relinking their programs with special libraries. Hence, these systems are neither secure nor transparent. In contrast, watchdogs purposefully involve the kernel to obtain both security and transparency.

Operating systems that permit the entire file system to reside outside the kernel, such as Mach Mach or Amoeba Amoeba overview are still not easily extendible. Although not kernel resident, these file systems are non-trivial programs unlikely to be modified by unsophisticated users. Because watchdogs are intended to handle isolated file system functions, they can be made simple enough so that even a novice programmer can master them.

The watchdog approach to directory management subsumes the unimplemented portal mechanism described in the original 4.2BSD UNIX documentation unix portals and is similar to Apollo's extended naming facility in the DOMAIN system.

Implementation

The demands of watchdogs are unique among all facilities in the UNIX operating system. Watchdogs must be associated with files; the kernel must be able to transparently effect communication between the process using a file and the watchdog controlling it; there must be some mechanism for the creation and management of watchdog processes; and finally, the system must be robust enough so that error conditions arising from the users' and kernel's reliance on potentially unreliable watchdogs do not cripple the system. The software framework to provide these facilities has several principal components:

- a new system call to link watchdogs to files,
- a message-based kernel/watchdog communication mechanism, and
- a system-wide *chief* watchdog process, responsible for starting new watchdogs and managing ones already running.

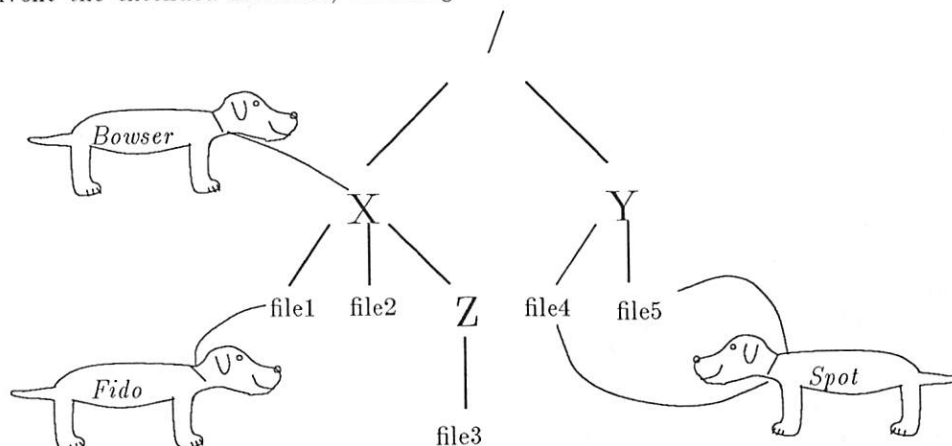


Figure 2 - Watchdogs In The File System Tree

This section describes the design and implementation of these components.

Binding the Watchdog to the File

The watchdog associated with a file is a characteristic of that file, much like its owner or protection mode. Consequently, the information belongs in the file's inode, where it can be modified only through secure system calls. The current implementation of 4.3BSD UNIX reserves 20 bytes in the inode for "future use." Since watchdogs are an experimental system, and since we did not want to make major changes to the inode subsystem (that is, reformat the disk), those 20 bytes are used to record the name of the executable image containing the watchdog. This imposes an annoying restriction on the length of a watchdog's name. This problem is circumvented by maintaining a public directory, `/wdogs`, containing symbolic links to real watchdogs scattered throughout the system. Access control to this public directory is, of course, managed by a watchdog.

To bind a watchdog to a file, a program makes the `wdlink()` system call:

```
int wdlink (char *file, char *watchdog);
```

A user program by the same name provides this interface from the shell. A user must be the file's owner (or root) to link in a watchdog. `Wdlink` with a null second argument unlinks any watchdog. The `ls` command has been augmented with yet another flag so that the name of an guarding watchdog can be seen on a directory listing.

```
% ls -lwd /wdogs
drwxrwxrwx 1 root 1024 Dec 25 11:43
      /wdogs <- /wdogs/wd_mgr
```

Since the name of the watchdog is kept in the file's inode, there is no limit to the number of files that may be guarded by a single watchdog. The implementation does however impose a limit (one) on the number of watchdogs that may guard a file. Having multiple watchdogs share responsibility for a single file might be useful in certain situations, but the added utility did not seem to warrant the extra complexity.

Kernel/Watchdog Communication: Watchdog Message Channels

Typically, communication between processes and the UNIX kernel is asymmetric and haphazard, relying on a large number of not very orthogonal system calls and an asynchronous signaling mechanism. The former allows processes to manipulate kernel facilities, while the latter provides for very simple message passing from the kernel to a process. Watchdogs require a richer form of communication. A process makes a system call requesting that some operation be performed on a file. The kernel, acting as a switch, routes that request either to the file system, normally, or to the watchdog, for a guarded operation. In the

latter case, the watchdog eventually responds to the kernel with results to be relayed back to the user, or with a response instructing the kernel to take action on the request.

The ability for the kernel and a process to treat one another as peers in a message-based communication environment exists in many other operating systems: mach, foundation for unix development accent, communication oriented network operating system basket, Task Communication in DEMOS but is not present in 4.3BSD UNIX. There were three possible ways to fill this void: implement general message passing between processes and the kernel, exploit an existing communication mechanism, or construct a message system specifically tailored for watchdogs.

The first approach would have been to implement a completely general message passing mechanism, similar to those present in other systems, replacing or augmenting the existing procedure call interface with a message-based one. Because this would involve fundamental changes to the system, because we wished to retain compatibility with the original 4.3 BSD system, and because the implications of such a major redesign effort were beyond the requirements of the project, this choice was rejected.

A second approach would have been to use existing socket code so that the kernel and a watchdog would communicate via standard UNIX sockets. Concern for performance kept us from adopting this approach. The characteristics of kernel/watchdog communication did not appear to require the generality provided by sockets.

The third choice, and the one taken, was to build a special-purpose message passing system. With this, optimizing for the anticipated message characteristics was possible. A watchdog communicates with the kernel via a Watchdog Message Channel (WMC). There is one WMC per watchdog. A WMC is created with the `createwmc()` call and referenced by a standard UNIX file descriptor using the operations read, write, close, etc. A new kernel descriptor type, `DTYPE_WDCHANNEL`, exists to support these operations on the channel. The kernel sends messages to the watchdog on a WMC to relay user requests and data for guarded operations, and the watchdog uses its channel to respond to those requests.

Messages contain a type field, a session identifier and the message contents. The session identifier permits the kernel and the watchdog to multiplex the activity of multiple files over a watchdog's single WMC. When a file is first opened, the kernel assigns to it the session identifier used in the open message. All subsequent messages between the kernel and watchdog referencing that opened file contain the same identifier.

Messages may be sent by either the kernel or the watchdog. For example, the kernel can send a message of type `Read_Request` to the watchdog. The message includes a file's session identifier, current offset,

and number of bytes to read. The watchdog may choose to respond with a simple message in the form of an acknowledgment, denying or deferring the operation, or it may respond with the actual data to be returned to the reading process.

A watchdog has dynamic control over where and how its address space is utilized on behalf of processes accessing files. In the case of a *Write_Request*, a watchdog's positive, non-deferring acknowledgment includes a pointer to where the written data should be placed in the watchdog's address space. For example, a process may write 40k bytes to a file in a single operation, but the watchdog may only be equipped to handle 4k bytes at a time. This restriction is enforced and the process's write system call always returns the number of bytes transferred into the watchdog's address space. Programs that do not properly check the return value of the write call, instead relying on the fact that the file system may impose no limit on the length of a written buffer, are technically in error and may be so exposed if they access a file guarded by a buffer-limited watchdog.

The structures required to associate a process with a watchdog during an open file session are illustrated in Figure 3. The solid lines indicate links needed to forward a process's access request on a guarded file to the actual watchdog, while the dashed lines reflect those needed to relay a watchdog's response back to the process. The user's per-process open file table points into the system-wide open file table. Watched entries in the system's open file table include a pointer into the watchdog session table. Each session table entry references the kernel's end of the WMC, where unread messages are kept. The WMC entry also contains a pointer to the process representing the watchdog. The watchdog's WMC indirectly references the entry in the WMC table

through the system's open file table. A message arriving from the watchdog includes the session identifier, which maps into the session table. Each session table entry has a back-pointer to the appropriate entry in the system's open file table, allowing watchdog responses to be returned to the correct process.

Managing Watchdog Processes

The kernel assumes a minimal role in the management of watchdog processes. It must be able to map from a watchdog's name onto that watchdog's WMC. This is done by remembering the device and inode number of a watchdog as it requests a WMC. Later, when a guarded file is opened, the device and inode number of the watchdog named in the opened file's inode are compared against those of the currently active WMCs. If a match is found, the watchdog is running and a new session is begun. Otherwise, the watchdog is not yet running and must be started.

Since a large part of the watchdog framework includes a kernel to process communication mechanism, this structure is used to allow a normal user-level process to manage all watchdogs, much like *inetd* manages network daemons. This manager process, known as the chief watchdog, communicates with the kernel through a normal WMC, having first announced itself as the chief by sending an *I-am-chief* message on the channel. On the opening of a guarded file for which no watchdog is currently running, the kernel sends a message to the chief indicating that a new watchdog should be created. The chief *execs* the requested watchdog which then opens its own WMC. The watchdog's first read on its new WMC returns the message describing the outstanding open request. If the chief is unable to start the watchdog, or if the

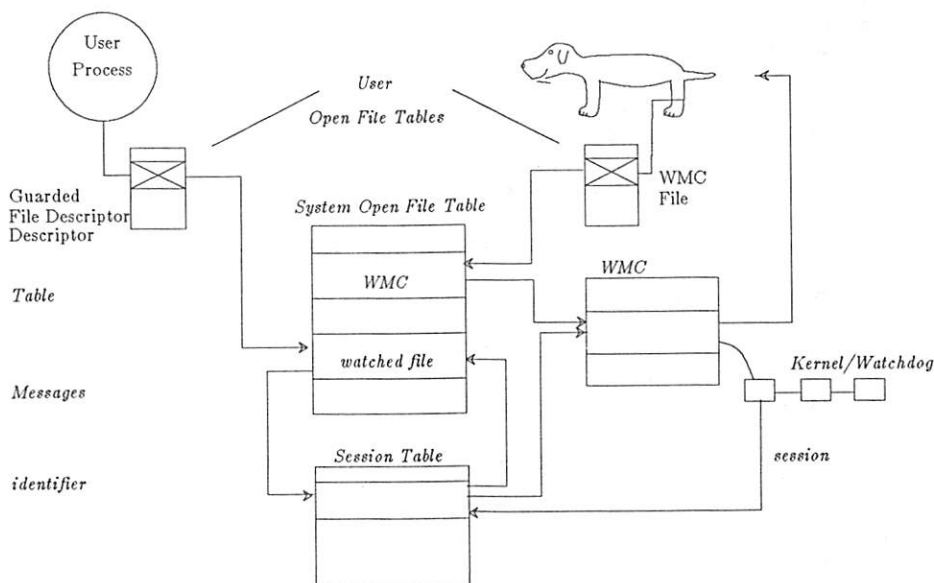


Figure 3 - Watchdog Data Paths

watchdog terminates abnormally, the chief notifies the kernel of the failure, returning to it the UNIX error code that should be relayed back to the opening process. Once running, a watchdog should not terminate until it has no active sessions. If a watchdog dies in the middle of a file session, processes referencing the file will, on their next access attempt, receive either an EOF or an error depending on the type of access method (i.e. *read* returns EOF, while *fstat* returns error).

Creating a new watchdog process on every open is expensive, at least two orders of magnitude slower than utilizing an existing watchdog. In situations where a watchdog is frequently used, the overhead of involving the chief and constructing a new process each time is unacceptable. By monitoring watchdog creation requests from the kernel, the chief may allow a heavily used watchdog to continue to run even though it might not be currently active. The chief's benevolence is based on the expectation that a frequently opened file is likely to be opened again in the near future. Thus, when the watchdog is required again there is no need to start a new process or even involve the chief.

Although the chief is normally responsible for unleashing most watchdogs, there is no restriction that prevents users from running them directly. Users can debug watchdogs by running them from within the shell or their favorite debugger. This is the normal mode for watchdog development.

Writing Watchdogs

The internal structure of a watchdog is similar to a server in a networked environment. The watchdog first creates a rendezvous port (WMC), and then listens patiently on that port awaiting requests. The

code fragment of Code Table 1 highlights the structure of a watchdog's implementation. Each of the functions in the switch takes care of one file operation and then responds back via the WMC with an acknowledgment directing the kernel to take an appropriate action on behalf of the process wishing to access the file.

A watchdog emulating the special file */dev/null* trivially implements the read and write functions as seen in Code Table 2.

Applications

This section describes several of the watchdogs that have been implemented since the system became operational. In general, these watchdogs extend the file system by either providing new functions not previously possible or replacing existing functions.

wdacl A file access controller that arbitrates over all opens of a file by verifying that the opener is mentioned in an access control list associated with the file being opened. Since each open request is accompanied by the name of the file to be opened, a single watchdog can be used to control access to many files.

wdcompact An on-the-fly compaction watchdog that allows files to be stored on the disk in compacted form, but viewed normally. Without watchdogs, this could only be done by manually inserting a pipe element to do the (un)compaction between every read and write.

wdbiff A "biff" watchdog that watches a user's mailbox for new mail and notifies the owner of its arrival. Without watchdogs, biffing can only be done through the combination of several ad-hoc mechanisms.

```

struct wdmmsg wdmmsg;
int cc;
int wmc = createwmc();
for (;;) {
    cc = read(wmc, &wdmmsg, sizeof(struct wdmmsg));
    if (cc != sizeof(struct wdmmsg)) {
        if (cc < 0)
            perror("read"), exit(-1);
        else
            abort(); /* should NEVER happen */
    }
    switch (wdmmsg.wm_type) {
        case WDMMSG_OPENREQ: do_open(wmc, &wdmmsg); break; /* open */
        case WDMMSG_READREQ: do_read(wmc, &wdmmsg); break; /* read */
        case WDMMSG_STATREQ: do_stat(wmc, &wdmmsg); break; /* stat */
        case WDMMSG_CLOSEREQ: do_close(wmc, &wdmmsg); break; /* close */
        /*
         * other guarded operations go here
         */
        default: /* should NEVER happen */
            abort();
    }
}

```

Code Table 1

wdview A directory watchdog that presents different views of the same directory depending on the user doing the query.

wdhfs A remote file system watchdog that guards a directory and provides heterogeneous remote access to files. Pathnames mentioning the guarded directory are resolved to the directory and then passed to the watchdog which remotely accesses the file. Information on the location of the remote file is obtained from a file similar to */etc/mstab*. The remote file system is part of the HCS project at the University of

Washington.

wddate A simple "date" watchdog that allows users to read the current time and date from a file. The file itself contains no data; all bytes emanate from a process reading the system clock.

The last example demonstrates that watchdogs can be used to provide a single, very simple, interface to serve many system functions. In this example, the UNIX *date* command has been eliminated.

```
do_read(wmc, wmsgp)
{
    int wmc;
    struct wmsg* wmsgp;

    /*
     * cast the message components into something short
     */
    struct wdrmsg *wi = (struct wdrmsg*)(wmsgp->w_un.wi_rw);
    /* read/write msg */
    struct wdackmsg *wa = (struct wdackmsg*)(wmsgp->w_un.wa_ack);
    /* ack msg */
    struct wdr wdr; /* request/reply ack including data */

    int error = 0;
    char maxbuf[BUFSIZ];

    wdr.wdr_sid = wmsgp->wm_sid; /* set return session id */

    if (wmsgp->wm_type == WMSG_READREQ) {
        wdr.wdr_len = 0;
        /* /dev/null read always returns 0 bytes */
        wdr.wdr_data = (char*)NULL;
        /*
         * acknowledge with data to return to process
         */
        error = ioctl(wmc, WDIOPDATA, &wdr); /* Put Data */
    } else {
        /* beware programs that don't check return values! */
        wdr.wdr_len = MAX(wi->wi_size, BUFSIZ);
        wdr.wdr_data = maxbuf;
        /*
         * acknowledge with reference to where data should be dumped
         */
        error = ioctl(wmc, WDIOPGDATA, &wdr); /* Get Data */
    }
    if (error) {
        /*
         * Data transfer failed. Must ack explicitly.
         */
        wmsgp->wm_type = WMSG_WDACK;
        wa->wa_status = error; /* returned to user process */
        write(wmc, wmsgp, sizeof(struct wmsg));
    }
}

do_write(wmc, wmsgp)
{
    int wmc;
    struct wmsg* wmsgp;

    do_read(wmc, wmsgp);
}
```

Code Table 2

Performance

For many users, the question of whether or not to use watchdogs will be decided by their performance. Although individual watchdogs have costs related to their complexity, all guarded operations impose a minimum overhead. This overhead is due to the cost of message passing and context switching incurred on each guarded operation. This section summarizes these basic costs for the most common file system operations: read, write and open.

Table 1 summarizes the read and write costs in terms of the average elapsed time to perform a single operation in the context of accessing a large file. First, we measured the cost of deferred operations where the only overhead is communication to and from the watchdog. From the standpoint of the process accessing the file, a deferred read or write adds about ten percent to the operation's elapsed time. Second, we looked at the time required for a process to read and write a guarded file when the watchdog does the actual reads and writes on behalf of that process. Two sets of figures are given for the reads: one with file read-ahead enabled and the other without. The reason that the elapsed time for guarded operations increases so much without read-ahead is that the watchdog processing for any one buffer can not be overlapped with the read-ahead of the next. Lastly, to mask the overhead of the disk access, we measured the cost of issuing a guarded read (or write) and having the watchdog return the data from an in-core buffer (essentially a cached file). The cost here is significantly less than any call involving the disk,

which shows that watchdogs are a viable means for caching frequently accessed files and that the overhead of the watchdog need not be excessive.

Table 2 compares three different types of open: a normal, unguarded open, a guarded open where the watchdog is already running, and a guarded open where the watchdog must be created. Creating a new watchdog process is costly because it involves a *fork()* and *exec()* by the chief watchdog. Such expense is acceptable for watchdogs that are not frequently used, but cannot be tolerated for those that may be invoked several times every second. The latter may be kept alive even during short periods of inactivity so that they can be referenced quickly. In the best case, opens involving these watchdogs are only three times slower than their unguarded counterparts. The worst case occurs when a file named by a relative path is opened and the kernel must determine the absolute pathname of the file for the watchdog. Since the name of the current working directory is not kept in the kernel, determining the full pathname of the file is expensive.

In situations where a file's contents must always be filtered before their examination (such as in compressed or encrypted files), watchdogs serve as a natural replacement for pipes. To compare the overhead of watchdogs to that of pipes, we conducted two experiments with pipes. The results are summarized in Table 3 (again, averaged over a large number of reads). In the first test, we sent a large amount of data through a pipe to measure the cost of reading one kilobyte from a sending process. This cost is very close to the cost of reading data from a cached file and suggests that watchdogs have data transfer times

Table 1. Average Elapsed Time for *read* and *write* (milliseconds)

	unguarded file	guarded (deferred)	guarded (not deferred)	guarded (watchdog cache)
read 1K (read-ahead)	5.064	5.572	5.348	2.880
read 1K (no read-ahead)	6.872	10.272	6.872	2.880
write 1K	8.092	8.168	8.304	2.916

Table 2. Average Elapsed Time to Open a File (milliseconds)

	unguarded file	guarded file (alive watchdog)	guarded file (dead watchdog)
open (absolute name)	3.070	9.591	108.0
open (relative name)	1.398	21.356	117.0

Table 3. Average Elapsed Time of Guarded Reads vs. Reads from a Pipe (milliseconds)

	data cached	data from file
read 1K, pipe	2.772	6.051
read 1K, guarded	2.880	5.348

similar to pipes. To test this hypothesis in a more realistic setting, we had a process read data from a file and write it to a pipe. We measured the time required for each read on the pipe and compared it to the time required to do a non-deferred read from a guarded file. Again we found that watchdogs have speed similar to that of pipes.

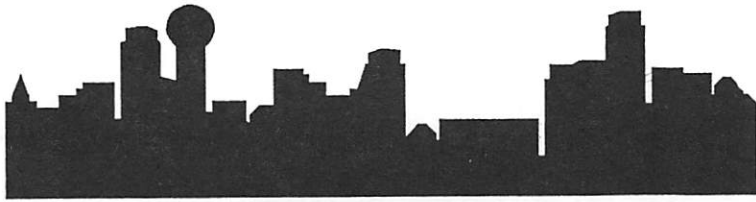
Watchdogs' performance can be improved in many ways. Opens can be enhanced by providing a more efficient determination of a file's full pathname. Several parts of the system rely on linear scans of kernel data structures to associate watchdogs with currently open files. These scans could be eliminated by using hash tables. Although the system is designed to allow page remapping between user and kernel space for transferring large blocks of data, this facility is currently unimplemented. Data is simply copied byte for byte between user and kernel. For reads and writes going through watchdogs to actual files, this means that any single byte of data will have to be copied across kernel boundaries three times, twice more than for normal file accesses, and once more than for pipes.

Conclusions

Watchdogs allow arbitrary redefinition of file system operations. Because watchdogs are implemented as user-level processes and interact with the kernel, they provide a means to transparently and securely change file system semantics. These qualities allow watchdogs to simplify the user's perspective of the file system by binding a file's contents to its operations. For example, the current date may be read directly from a file, the line printer daemon (*/usr/lib/lpd*) may guard */dev/printer* to arbitrate access to the hardware, and a mailbox may automatically "biff" its owner upon receipt of new mail. We have demonstrated that watchdogs can incur low system overhead and exhibit good performance. Nevertheless, we expect that performance can be further improved as the system matures. We believe that the main concept demonstrated by watchdogs, namely the ability to easily redefine single operating system functions (as opposed to entire subsystems) at the user level, is an important one and should be included in modern operating systems.

Acknowledgements

We'd like to thank Ed Lazowska, Hank Levy, David Notkin and Ellen Ratajak for their helpful comments on earlier drafts of this paper.



Invoking System Calls from Within the UNIX Kernel

Mike Mitchell, Kent Moat, Tom Truscott,
and Bob Warren
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, North Carolina 27709
(919) 541-7005
mcnc!rti!{mcm,trt,rbw}

ABSTRACT

We have added a `ksyscall` routine to the UNIX kernel which permits calling most UNIX system calls from within the kernel. This provides considerable power and flexibility to the UNIX kernel hacker. It brings kernel hacking within the reach of programmers who are still trying to figure out the difference between `iput` and `ifree`. It is portable and is currently running on about twenty different UNIX implementations. Code written using `ksyscall` is itself portable to other kernels.

This paper describes the motivation for writing `ksyscall` and gives examples of its use. It discusses the implementation and considers issues of portability, efficiency, and effective use of `ksyscall`. It concludes by outlining the role of `ksyscall` in hierarchical system construction and by comparing this approach with possible alternatives.

Motivation (Through the Looking Glass)

When a UNIX process issues a system call, it passes through the protection interface and into a strange and unfriendly kernel. This is a world of dancing inodes and other objects that must be carefully juggled or they will fall to the floor. As the kernel becomes bigger and better, the juggling becomes harder and faster! What happened to the friendly system call interface? Why don't system calls call other system calls to do their job, in the same way that library routines call other library routines? One could then build on the work of others, rather than stepping on it.

We originally wrote `ksyscall` to provide efficient yet portable support for the FREEDOMNET distributed computing system, which provides remote file/device access and remote program execution.¹ `Ksyscall` allows the FREEDOMNET "device driver" to access configuration files and varying network interfaces in a manner resembling normal UNIX programming, rather than via non-standard (and undocumented) internal routines. This is particularly helpful as most ports have been on systems with binary-only UNIX distributions. Of course, the benefits of `ksyscall` extend well beyond this original goal.

Some Examples

If the kernel expands the tilde ("~") character the way the C-shell and many other programs do, *any* program can open `~dmr/plastercast` and get the

expected results. The kernel can simply open and read `/etc/passwd` to find `dmr`'s entry:

```
fd = kopen("/etc/passwd", 0);  
while ((n = kread(fd, buf, sizeof(buf))) > 0)  
    . . .  
kclose(fd);
```

One no longer has to call `namei`, `iget`, `iput`, `irele`, `bread` and others in mystical combinations to get at files. The `kopen` and other "k" interface routines themselves call `ksyscall`. For example, `kopen` is simply:

```
int      kopen(name, mode, creatmode)  
char     *name;  
int      mode, creatmode;  
{  
    return(ksyscall(open, &name,  
                     0x10000003L, strlen(name)+1));  
}
```

The details of `ksyscall` will be explained later, but it should already be obvious why we prefer to use the interface routines.

The tilde expansion feature was implemented and tested in a few hours. Unfortunately, looking up users in `/etc/passwd` does not work properly on "Yellow Pages" systems, and moving "Yellow Pages" into the kernel seemed like a bad idea. We decided instead to write a user-mode server daemon which the kernel consults when it gets a pathname that might require translation. This reimplementaion also took only a few hours, most of which was spent coding the

server. The client code was first written and tested in a user-mode program, then moved into the kernel by changing `socket` to `ksocket` and so on.

Does tilde processing belong in the kernel? Our experience suggests so, and it certainly seems cleaner than the current mostly compatible implementations in a dozen or so “important” programs. With a common pathname translator in place other useful transformations beyond tilde expansion might come to mind. It is probable that the perceived difficulty of kernel hacking has led to the current situation in which such ideas are stifled. Removing this artificial complexity may result in finding new answers to old questions.

As an exercise we rewrote the 4.3BSD kernel core routine (which writes out “core” files) using `ksyscall`. The 14 lines of code in Code Table 1 replace 37 lines in the original.

We claim the `ksyscall` version is much easier to understand. The trade secret status of `core` precludes listing it, so instead let’s have a pop quiz. Quickly now (answers at end):

- How many arguments does the `write` system call have? _____
- How many arguments does the 4.3BSD `rdwri` routine have? _____
- Most “vnode” (NFS) systems lack `rdwri` and instead use: _____
- (extra credit) Why did 4.3BSD add a second argument to `maknode`?

We have used `ksyscall` for doing non-trivial kernel work as well. For example, we used it to implement a system call tracer which writes system call traces and usage statistics to files and/or user’s terminals (using `kwrite`, of course). The tracer originally was intended for debugging and tuning FREEDOMNET but has turned out to be even more useful for debugging and tuning application programs, especially those involving daemon processes. An implementation at the “input” level would not only be more difficult and less portable, it would also be much

```
if ((fd = kopen("core", O_WRONLY|O_CREAT, 0644)) < 0)
    return(0);
if (kfststat(fd, &sb) == 0
    && (sb.st_mode & IFMT) == IFREG && sb.st_nlink == 1
    && kfsttruncate(fd, 0L) == 0
    && kwrite(fd, &u, ctob(UPAGES))
        == ctob(UPAGES)                                /* u area */
    && kwrite(fd, ctob(dptov(u.u_procp, 0)), ctob(u.u_dsize))
        == ctob(u.u_dsize)                             /* data */
    && kwrite(fd, ctob(sptov(u.u_procp, u.u_ssize-1)), ctob(u.u_ssize))
        == ctob(u.u_ssize))                             /* stack */
    u.u_acflag |= ACORE;
kclose(fd);
return(u.u_acflag & ACORE);
```

Code Table 1

less trustworthy. This is particularly true when interruptible terminal I/O is involved. Our confidence in the correctness of the system call tracer derives in large part from our confidence in `ksyscall` and in the underlying UNIX system calls themselves.

And of course we used `ksyscall` when implementing FREEDOMNET. We avoid it in frequently executed routines but find it a boon when implementing more esoteric functions. It is also useful when experimenting with new features as they can be implemented with relative ease. We suspect that some ideas would otherwise never have been thought of (or at least not been considered seriously).

These examples will reappear in the discussion of “system call layers.”

Implementation (Taming the Wild System Call)

If using system calls from within the kernel is a good idea, then why is `ksyscall` needed at all? Why not simply call the internal kernel routines for `read`, `open`, etc. directly? One reason is that these kernel routines are not themselves re-entrant. `Ksyscall` ensures re-entrancy by saving and restoring all volatile state information. A second, more fundamental reason is that the kernel system call routines expect their arguments to be in user address space, not kernel address space. Hence, `ksyscall` must arrange that the arguments for the kernel routines be located appropriately.

This section discusses how `ksyscall` does these jobs. It describes kernel internals only as they apply to the operation of the `ksyscall` mechanism. For a more complete description of kernel internals (for System V but largely applicable to Berkeley-based systems as well) see Bach.²

The User Area

`Ksyscall` manipulates or accesses information stored in the *user area* of a process. The user area or *u area* is a kernel data structure associated with each user process which contains information describing a process which needs to be available only when the

process is running. In particular, `ksyscall` manipulates or accesses the following fields of the `u` area structure:

- `u_arg` array of arguments to current system call
- `u_ap` pointer to argument list
- `u_rval1, u_rval2` system call return values
- `u_errno` return error code
- `u_qsave` for non-local gotos on interrupt
- `u_procp` pointer to process structure associated with this `u` area

These fields will be referred to below.

Ksyscall Calling Sequence

The `ksyscall` mechanism is invoked as follows:

```
rc = ksyscall(callp, argp, vec
               [, size, ...])
```

where the arguments are

- `callp` pointer to kernel routine for system call
- `argp` array of arguments for system call
- `vec` bit vector indicating number of arguments, etc.

- `[size, ...]` size of each argument copied in/out
- `rc` return code for system call execution

Note the variable number of arguments. This will be explained below.

First Cut Implementation

Figure 1 gives a first cut implementation of the `ksyscall` mechanism. The first step is to save all

```

Loop over arguments
  Save argument (u_arg[i], argp[i])
  Load in passed argument
Save, then set up argument pointer (u_ap)
Save, then clear return values, errno
  (u_rval1, u_rval2, u_error)
Make the system call (callp)
If there was an error
  return_code = -1 (rc)
Else
  return_code = return from syscall
Restore return values, errno
  (u_rval1, u_rval2, u_error)
Restore argument pointer (u_ap)
Loop over arguments
  Restore argument (u_arg[i], argp[i])
Return return_code(rc)
```

Figure 1. Straightforward `ksyscall` Implementation

relevant state of the user-initiated system call so that the desired system call can be executed internal to the

kernel. This is important to ensure re-entrancy. Next, the desired system call is invoked. The return code is passed back to the invoker in `rc`. Finally, the user-initiated system call state is restored.

While this seems very straightforward, there are a number of problems with this approach as it stands. Signals are not handled at all. Without proper signal handling, `ksyscall` would not work properly. More importantly, this approach assumes all arguments to the desired system call are in the kernel's address space. This is rarely the case. Both of these considerations add to the complexity of the basic mechanism.

Dealing with Signals

`Ksyscall` needs to handle signals (such as that generated when the user types her interrupt character) for two important reasons. First, any signals pending on a user-level initiated system call would interfere with the execution of the system call invoked by `ksyscall`. Second, if the invoked system call is interrupted, then `ksyscall` needs to get control to perform clean up actions before returning control to the kernel. Figure 2 shows the changes needed to deal with signals.

```

Loop over arguments
  Save argument (u_arg[i], argp[i])
  Load in passed argument
Save, then set up argument pointer (u_ap)
Save, then clear return values, errno
  (u_rval1, u_rval2, u_error)
Save, then clear signal information (u_procp->p_sig)
Save signal setjmp buffer (u_qsave)
Protect system call with setjmp (u_qsave)

Make the system call (callp)

If there was an error
  return_code = -1 (rc)
Else
  return_code = return from syscall

Restore setjmp buffer (u_qsave)
OR-in saved signal information (u_procp->p_sig)
Restore return values, errno
  (u_rval1, u_rval2, u_error)
Restore argument pointer (u_ap)
Loop over arguments
  Restore argument (u_arg[i], argp[i])

Return return_code(rc)
```

Figure 2. Handling Signals in `ksyscall`

The kernel keeps a bit vector of pending signals in the process table (`psig`). This table is accessible to `ksyscall` since it is pointed to by `u_procp`. Hence, `ksyscall` first saves any existing signals and then clears them. In addition, in kernel implementations which maintain the current signal (`u_procp-`

>p_cursig), this must also be saved and cleared.

The kernel normally guards the execution of a system call with a `setjmp` call. The `setjmp` buffer is set up by the kernel in its system call dispatch routine. If an interrupt occurs, a `longjmp` occurs to clean up or cause the system call to be restarted. `Ksyscall` must save this `setjmp` buffer (`u_qsave`) and set up its own to do similar clean up and error reporting should an interrupt occur.

Once the `ksyscall` has been executed, the sig-

```
Loop over arguments
  Save argument (u_arg[i], argp[i])
  If it should be in user space (vec)
    Remember size, offset (size)
Allocate necessary user space
Loop over arguments (u_arg[i])
  If it should be in user space (vec)
    Load in user space address
    Copyout from passed argument if necessary
  Else
    Load in passed argument
Save, then set up argument pointer (u_ap)
Save, then clear return values, errno
    (u_rval1, u_rval2, u_error)
Save, then clear signal information
    (u_procp->p_sig)
Save signal setjmp buffer (u_qsave)
Protect system call with setjmp (u_qsave)

Make the system call (callp)

If there was an error
  return_code = -1 (rc)
Else
  return_code = return from syscall

Restore setjmp buffer (u_qsave)
OR-in saved signal information (u_procp->p_sig)
Restore return values, errno
    (u_rval1, u_rval2, u_error)
Restore argument pointer (u_ap)
Loop over arguments
  If it should be moved from user space (vec)
    Copyin to passed argument
  Restore argument (u_arg[i], argp[i])
Free any user space allocated

Return return_code (rc)
```

Figure 3. Copying Arguments In/Out in `ksyscall`

nal information is restored. This is done in such a way as to also preserve any new signals which may have occurred during `ksyscall` execution.

Dealing with Arguments in User Space

Most kernel system calls expect some or all of their arguments to reside in user address space, not kernel address space. This is the normal case for user programs requesting kernel services. In the case of system calls executed from within the kernel, the arguments must still be passed in from user space. `Ksyscall` must arrange for the appropriate arguments to be copied out to user space before the system call is invoked, and any changed arguments to be copied back in after execution.

Figure 3 shows the changes necessary to make sure arguments are copied out and copied in appropriately. The `vec` argument to `ksyscall` is checked to determine which arguments must be copied out to user space so the kernel system call can access them, which arguments must be copied in from user space after being set by the kernel system call, and which arguments must be both copied out and copied in. The `sizes` of those which must be copied out of/into user space are totaled. Sufficient user space memory is allocated, and the appropriate arguments are copied out to user space. For portability reasons, we chose to allocate user memory by moving the “break.” This is done by calling the `brk` system call. Since `brk` has only one argument, a simple in-line form of `ksyscall` is used to avoid unnecessary recursion. The argument list for the internal system call is set properly to reflect the new addresses of the arguments. After the system call is executed, all arguments which are changed in user space are copied into kernel space. Finally, the allocated user space memory is freed.

Other Considerations

The above implementation still glosses over a number of issues. For example, on System V-based systems there is an additional field in the `u` area which must be saved and restored, the directory pointer (`u_dirp`). Some kernels do in-line editing of the `setjmp` call which necessitates some minor changes to `ksyscall`. And, since kernel stack space is at a premium on some systems, changes may be required to minimize stack usage.

Portability

If we had only wanted a `ksyscall` mechanism for one particular operating system the implementation of `ksyscall` could have been simplified greatly. For example, the method of memory allocation for 4.3BSD could have been done instead by calling the routines `swpexpand` and `expand` directly. However, the goal was to be portable over as many different implementations of UNIX as possible, especially for FREEDOMNET ports. We have ported `ksyscall` to about 20 UNIX variants, including System V Release 2 and 3, Berkeley and Berkeley-based systems, and hybrids of System V and Berkeley.

Ksyscall Programming Considerations

The following considerations must be kept in mind when using `ksyscall`:

1. A mistake in kernel programming causes a system crash, not a core dump.
2. Not all system calls can be performed within the kernel. For example, `signal` can be used to ignore or take the default action on a given signal but cannot specify an in-kernel signal handling routine.
3. Kernel stack space is limited and running out of it causes a system crash. Use `malloc` (or whatever your kernel provides) to allocate needed buffer space. Do *not* simply declare such buffer space to be `static` (why?). Avoid deeply nested subroutine calls.
4. Avoid global variables, as they are common to *all* processes.

"System Call Layers" (Ring Around the UNIX)

At one level the in-kernel system call facility is simply a subroutine library which makes kernel pro-

facility allows building hierarchical "system call layers." For example, it is instructive to take a fresh look at the system call "tracing layer" (Figure 4).

A process using the tracing layer sees an enhanced system call interface in which each system call functions as before (by calling the original system call code), and has the side effect of recording its invocation (by using `ksyscall` to invoke various other system calls).

The FREEDOMNET system call "distribution layer" functions similarly (Figure 5). The distribution layer examines the system calls to see if remote objects are being referenced. If not, the system call is passed unchanged to the local kernel. If so, an appropriate remote system call is constructed and passed to the remote machine where the system call is executed by the remote kernel. The result is passed back to the local machine and relayed back to the invoking program. `ksyscall` makes possible the use of the normal networking capabilities from within the kernel to achieve this.

Contrast this approach with the "inode distribution layer" approach used by Sun's Network File System,³ and AT&T's Remote File Sharing.⁴ The distribution capabilities of these systems come at the cost of major kernel source changes, lack of portability, and limitations on the scope of the distributed system. A distribution layer implemented at the inode level tends to become blurry since it is so tempting to make changes unrelated to the distribution function throughout the kernel. A distribution layer implemented at the system call level is cleaner since it occurs at a well-defined interface into the kernel.

Other layers come to mind, such as a "fault-tolerant layer" which replicates files (and processes),⁵ a layer which performs data encryption,⁶ or a layer which balances the load among several processors. And, of course, the layers are stackable so that one can trace a program which is accessing remote files and so on (each layer should call the next lower layer, or at least be careful to avoid unintended recursion).

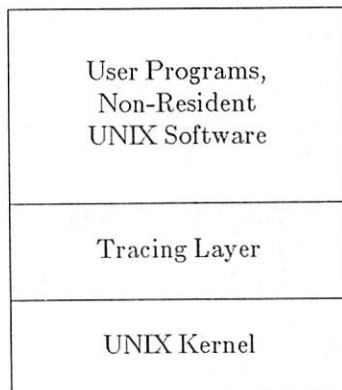


Figure 4. System Call Tracing Layer

gramming easier, faster, safer, etc., but adds nothing new to what UNIX application programmers have had all along. At another level, though, the system call

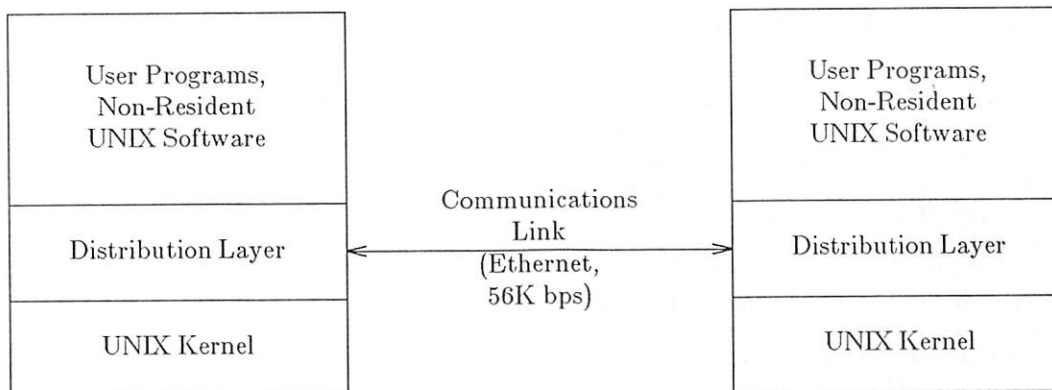


Figure 5. System Call Distribution Layer

Related Work

The System V Release 3 stackable line disciplines (STREAMS)⁷ may be thought of as a special case of system call layers. By concentrating on those aspects unique to stream I/O, STREAMS provides convenient, powerful, and efficient support for layered network protocols. However, it is not clear that a more general "stackable system calls" implementation would be any less convenient, powerful, or efficient.

Shared libraries, only now becoming widespread on UNIX systems, can in principle be used to provide "library layers." This is significantly more general than layered system calls and has the advantage of being implemented in user space. Unfortunately, some implementations of shared libraries do not appear to be stackable. Also, user-mode routines cannot perform privileged operations as can kernel-mode routines.

Protection Rings⁸ provide, in effect, set-user-ID subroutine calls and thus might be a usable alternative to `ksyscall`. There remains the question of whether the kernel itself can call on these routines to do its job. Also, protection rings blur the kernel/non-kernel distinction and can so strongly affect the operating system implementation that comparison with the `ksyscall` approach may be meaningless.

Whatever the merits of the above, their implementation involves substantial kernel changes and, thus, do not meet our portability requirements.

The Newcastle Connection (or UNIX United),⁹ on which FREEDOMNET is based, provided an out-of-kernel implementation of a distribution layer. Indeed, UNIX United grew out of work based on a layered approach to fault-tolerance, load balancing, and multi-level security.

Conclusions

The utility of the `ksyscall` mechanism is demonstrated by the examples of the tilde server and the tracing and distribution layers (FREEDOMNET). We are using it in work on fault-tolerant and load balancing layers.

So long as UNIX kernels continue their breathtaking growth, there will be a need to control that growth. `Ksyscall` and other closure mechanisms can help structure and possibly even slow the growth by allowing the kernel programmer to use the most powerful tools available rather than more primitive building blocks. They also focus attention on the system call interface, which after all is the essence of UNIX. That interface is not perfect, nor can it be, so it must undergo continual scrutiny for possible improvements, corrections, and simplifications.

UNIX and its system call interface have brought joy and enlightenment to many who formerly had to fight their way through baroque "operating system

services." `Ksyscall` may bring some measure of similar joy to those who labor in the kernel. Perhaps future kernels will have a more layered design, possibly obviating the need for `ksyscall` per se. We hope, and suspect, that the most interesting uses of this approach are yet to come.

References

1. Bob Warren, Tom Truscott, Kent Moat, and Mike Mitchell, "Distributed Computing Using FREEDOMNET in a Heterogeneous UNIX Environment," *Proceedings of the 1987 Unix Forum Conference*, pp. 115-126 (January 20-23, 1987).
2. Maurice J. Bach, *The Design of the UNIX Operating System*. 1987.
3. Dan Walsh, et. al., "Overview of the Sun Network File System," *USENIX Association Proceedings*, pp. 117-124 (January 23-25, 1985).
4. Andrew P. Rifkin, et. al., "Remote File Sharing Architectural Overview," *USENIX Association Proceedings*, pp. 248-259 (June 11-13, 1986).
5. T. Anderson, P. A. Lee, and S. K. Shrivastava, "A Model of Recoverability in Multi-level Systems," *IEEE Transactions on Software Engineering* SE-4(6) pp. 486-494 (1978).
6. John Rushby and Brian Randell, "A Distributed Secure System," *Computer*, pp. 55-67 (July, 1983).
7. Dennis M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal* 63(8) pp. 1897-1910 (October, 1984).
8. R. M. Graham, "Protection in an Information Processing Utility," *Communications of the ACM* 11(5) pp. 365-369 (May, 1968).
9. D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!," *Software — Practice and Experience* 12 pp. 1147-1162 (1982).

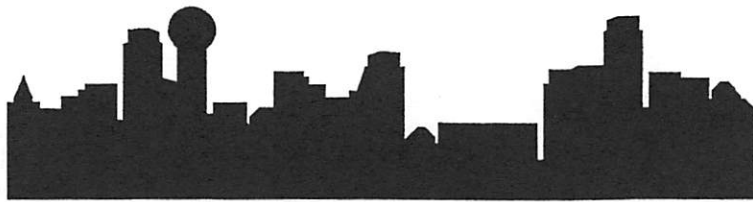
Quiz Answers

`write` has 3 arguments.

`rdwri` has 7 arguments.

Vnode systems use `vn_rdwr` instead of `rdwri`. It has 8 arguments.

The second argument to `maknode` is to force the new file's inode to be written to disk before its directory entry.



USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

Graham Hamilton & Daniel S. Conde
Digital Equipment Corporation
100 Hamilton Avenue
Palo Alto, CA 94301

An Experimental Symmetric Multiprocessor Ultrix Kernel

ABSTRACT

Multiprocessor systems are becoming increasingly common, but there have still been very few full scale Unix multiprocessor ports. We describe the goals, design, implementation and performance of one experimental symmetric multiprocessor version of the Ultrix kernel.

Introduction

Multiprocessor hardware is becoming increasingly common. The easiest way of providing Unix support for these machines is asymmetrical multiprocessing whereby user code may run on any processor, but all significant kernel calls must execute on the master processor. This approach for asymmetric Unix was pioneered at Purdue in 1981 [Goble81] and has successfully been used on a number of commercial systems [Finger85] [Probert86]. Unfortunately, it suffers from some inherent limitations. The master processor constitutes an obvious bottleneck on kernel performance, limiting the total amount of system activity that may occur, regardless of how many processors are added. Additionally, because all significant system calls must force a process switch to the master, i/o intensive code will suffer from needing to pay the price of an extra process switch whenever a system call is performed on a slave processor. These factors lead to the asymmetric system providing increasingly inadequate performance on systems with more than two processors.

The alternative approach is to modify the kernel so as to permit several processors to be executing significant kernel code simultaneously. We shall refer to this as Symmetric Multi-Processing or SMP. Depending on the implementation, this approach holds the possibility of scaling effectively to ten or more processors [Beck84] [Test86].

Goals

Our principal goal was to build an SMP kernel that could effectively exploit a ten processor system under a normal mixed work-load. We used a three cpu machine (a modified VAX 8300) as our development machine, but targeted our design to exploit a larger number of processors. (Ten was a convenient arbitrary round number to chose.)

Since Unix processes may easily spend 20% of

their time in the kernel, this goal implies that there must be a considerable amount of concurrent execution occurring within the kernel and that there must be comparatively little overhead involved in synchronization between different processors executing kernel functions.

A secondary goal of almost equal importance to performance was to develop a consistent design and implementation strategy that would make the SMP kernel relatively easy to debug, maintain and modify. Highly concurrent systems are notoriously prone to subtle timing bugs and to various forms of deadlocks. The interrelationships of the different components must be carefully maintained if the overall system is to continue to function. We aimed to make it relatively straightforward to codify the main concurrency relationships and to make it possible to validate them at run time, thereby making it relatively easy to ensure that changes to the system did not break any existing concurrency properties.

Minor goals included the desire to be able to incorporate existing device drivers into the SMP kernel without any source code changes. (However, such device drivers may not achieve such high performance as device drivers that have been modified to exploit concurrency.) As a final goal we wanted to be able to run the SMP kernel on a uniprocessor with only a minor (say 3%) performance degradation over a standard uniprocessor kernel.

Since this was an experimental prototype rather than a product, we also laid some restrictions on ourselves. Firstly we aimed to avoid making major structural changes to the kernel, preferring to work within the existing structures wherever possible and then to establish by experience which areas really needed restructuring. Secondly, within this constraint we aimed to try and obtain as much concurrency as possible within the kernel so that we could establish what the true limits were and what the natural bottlenecks

were. Thus we aimed to learn as much as possible about concurrency in the kernel while doing as little damage as possible.

Locking primitives.

Spinlocks and mutexes.

The main issue in making the Unix kernel multiprocessor safe is providing locking to prevent unfortunate interactions between different processors operating in the kernel. Sometimes locks can be thought of as protecting individual data structures, but it is often more useful to think of them protecting relationships or "invariants" such as "when this lock is free then X is the head of a well formed doubly linked list". Such invariants are often somewhat nebulous entities involving relationships between scattered data structures, e.g. such as "when this segment lock is free, all the segment PTE's describe valid entries in the cmap and none of the valid pages are on the free list and".

We used a combination of spinlocks and mutexes to provide locking within our SMP kernel. Briefly, a spinlock is a way of resolving conflicts between processors in a multiprocessor system whereby one processor repeatedly executes some form of interlocked instruction until it succeeds in gaining access to a resource. Conversely, a mutex is implemented by having one process try to grab a lock and if it fails having it sleep until the lock becomes available. The standard Ultrix kernel already uses informal mutexes, using BUSY and WANTED bits, together with sleeps and wakeups, to lock a number of resources such as inodes or buffers. Note that a processor remains "busy" while waiting to grab a spinlock, but switches to another process while waiting to claim a mutex.

Spinlocks are used whenever a data structure needs to be locked from an interrupt handler, since an interrupt handler cannot afford to risk sleeping. Additionally, it is desirable to use a spinlock whenever the lock is normally held for a shorter time than would be taken by a sleep and wakeup. However, it is never desirable to hold a spinlock across a sleep, as this runs the risk of a lengthy freezing of the other CPU's in the system. Thus mutexes must be used in all situations where an invariant must be held across a sleep.

Unlike some other multiprocessor Unix projects [Bach84] [Beck84], our design almost exclusively uses spinlocks to resolve conflicts that grow out of multiprocessor concurrency and merely used mutexes to replace the informal BUSY and WANTED locking that already existed in the kernel. This arose from the observation that any locking that was necessary across sleeps was already implemented using BUSY and WANTED flags in the standard kernel and that the bulk of the new, multiprocessor locking required was either of very short duration, or required access from interrupt handlers.

We provided the obvious locking and unlocking

primitives as macros. In non-debug mode the spinlock operations are compiled inline into three simple instructions each for the normal case when there is no conflict.

In combining the traditional sleep and wakeup mechanism with spinlocks and mutexes, we found it useful to provide two special macros which atomically combine a sleep with the release of either a mutex or a spin-lock. Specifically these macros guarantee that no other process will be able to execute a wakeup between the current process releasing the lock and actually sleeping. This guarantee is necessary to cope with the race condition between a process sleeping on a particular event and another process waking up on that event. If process B issues the wakeup after A decides a sleep is necessary, but before A actually sleeps, then A will miss the event and may sleep forever. To avoid this race condition it is necessary to use a lock to ensure that only one process can be accessing the relevant data structures at a time and then to use the combined sleep+unlock operation to guarantee that the race doesn't re-emerge between A releasing the lock and actually sleeping. The sleep+unlock operation is implemented by first acquiring the global scheduling spin-lock (thereby inhibiting any wakeups), then freeing the lock, then performing the actual sleep.

Cross processor interrupts

Spinlocks and mutexes are our main forms of synchronization. However it is occasionally necessary for one CPU to request that another CPU perform some specific service for it. We use the VAX's cross processor interrupts for this purpose. These requests include forcing a CPU to reschedule when a higher priority process becomes runnable, forcing a multiprocessor panic, printing high priority console output and forcing a multiprocessor translation buffer flush when the system page tables are updated. Additionally we use cross-processor interrupts in our page daemon's page invalidation code to ensure synchronization amongst all those CPUs which are running user processes with access to the soon-to-be-invalid page.

Some of these cross-processor interrupts occur from extremely low level code, which may hold important spinlocks. If one of the target CPUs is already spinning at high IPL on one of these spinlocks, then we risk a deadlock.

We solved this problem by adding extra code to our spinlock loop. Our main spinlock loop is called as a subroutine from the SGRAB macro, after a lock conflict has been detected. Each time round this loop we check our cross-processor interrupt flags and process any high priority requests. This means that a processor that is spinning for a lock will still service cross-processor requests even though the direct hardware interrupt may be blocked out by the current IPL level.

Locking granularity

It would be possible to implement a system with only a very small number of locks, with each lock controlling access to all the data structures of a major sub-system. For example there might be single lock for all virtual memory resources. This approach is superficially easy to implement, but is likely to lead to excessive contention for locks in the kernel.

We chose to try to maximize concurrency in the kernel by locking on a fine grain so that different processors could execute in the same code provided they were accessing different data structures. For example, we chose to have a spinlock for each tty structure, rather than having a global lock for all the tty code. (We still of course needed to have global locks for mapping tables and resource pools, e.g. the internet pcb list or the tty cfree list.)

Given the very large number of distinct data types in the kernel, there still remains the issue of deciding at precisely which granularity to lock. For example, in the network code, one might chose to independently lock each socket, protocol control block, mbuf, etc, leading to an absolute torrent of locks. This would both damage performance (by causing an excessive number of lock and unlock operations) and considerably complicate the code, which must acquire and release all these locks in the correct order, at the correct times.

We chose to apply a "locality of locking" rule of thumb to our selection of locks. If you would typically need to lock data structure A whenever you needed to update B or C, then you might as well not have independent locks for B and C, but rather define them as being locked by A. For example, in the network code we defined a lock for each socket which controlled access to the socket, the protocol control blocks the data mbuf chains, etc, on the grounds that locking them independently appeared to generate more problems than concurrency. Similarly, in the virtual memory code, we defined a lock for each logical segment, which controlled the segment data in the process or text or shared memory structure, and also the page table entries for the segment, the cmap entries for any allocated pages, the swap information, etc. Thus there is neither a lock for each cmap entry nor a lock for the cmap as a whole, but rather each cmap entry is locked either via a segment lock or via the free list lock, depending on its state.

Problems with the lock granularity

Unfortunately, even moderately fine grain locking cannot be achieved without some pain. Having multiple locks is occasionally troublesome to manage. For example, the pagedaemon has to go to some work to figure out which locks it needs to be dealing with. Additionally, the need to acquire multiple locks for some actions does increase the risk of deadlock. However, the additional concurrency that can be obtained seems to justify these difficulties.

There are two main cases where we failed to

obtain as much concurrency as we had originally hoped. We have a single lock for the file system buffer pool. We had originally planned to have multiple locks - one for each hash chain and one for each freelist. However we found it unacceptably complicated to try to manage these two sets of locks on the same objects, within the current buffer pool design. Instead, it appears more effective to try to minimize the amount of time that is spent in the buffer pool management code.

Similarly, we had hoped to have multiple locks controlling the text page chaining associated with the virtual memory freelist. Again, this effort appeared to cause more complexity than concurrency. Instead we made some minor rearrangements to the page allocation and deallocation code which reduced the amount of time the lock needed to be held.

However, as we describe in section 6.3, neither of these limitations seem likely to prevent our design from scaling beyond its goal of ten processors.

Affinity

So far we have assumed that all the CPUs have identical access to all the i/o devices and that all code can be converted to be SMP safe. In practice, some devices (notably the VAX console) may only be accessible from one CPU and we may not wish to convert all our device drivers or network protocols to be safe under SMP.

Each of our processes has an affinity mask, specifying which processors it is prepared to run on. The scheduling loop on each CPU checks this affinity mask and only runs processes which have the correct affinity. Macros permit a process to change its affinity (forcing a CPU switch if necessary). These macros impose very little overhead when there is no actual change of affinity.

We associate an affinity mask with each device driver and with each network protocol family. When a socket or tty is created, it is labelled with the affinity mask belonging to its device or protocol family. When we call into a device or invoke an operation on a socket or tty, we first invoke a macro to appropriately redefine our processes affinity and we also restore our previous affinity after the call. These macros ensure that a device or protocol family is only entered on a CPU which is in its affinity mask. It is also necessary to ensure that interrupts from such devices are only directed to appropriate CPUs. (This is potentially a serious problem if it is not possible to individually direct interrupts for each distinct class of i/o devices.) In the case of protocol families, we use fully symmetric network device drivers, but use cross processor interrupts to force incoming protocol handling on the correct processor.

Thus SMP devices and protocol families use locks for synchronization, whereas non-SMP devices and protocols are only ever invoked on a single CPU. This requires care in the "generic" tty and socket

code which may be invoked on both SMP and non-SMP objects. To avoid problems with locks, we use special socket and tty locking macros, which only perform locking actions if they are dealing with SMP objects.

Amongst devices, we have modified our disk devices drivers to be SMP safe and have used both SMP safe and non-SMP safe tty drivers. Amongst protocol families, we currently run SMP safe versions of the Arpa protocols, and non-SMP versions of the LAT and DECnet protocols.

The debug strategy

Within each lock and each process we maintain a lock log of the most recent lock operations, the relevant processes and the pc's at which they released the lock. This additional debug code is completely controlled by compile time flags and imposes no overhead on non-debug code. We also include a number of checks in this debug code to ensure that our locks are being used consistently (see 5.4 below).

One of the greatest risks of an SMP system is that someone will update a data structure without acquiring the correct lock. Major subsystems such as the socket code or the virtual memory code may have many subroutines that must be called with certain locks held, but other subroutines that require no locks to be held. It is extremely easy to lose track of which locks must be held on which procedure calls.

To codify our use of locks, we defined macros SASSERT and MUASSERT to be used whenever a procedure expected to be called with some spinlocks or mutexes already held. In debug mode, these macros check that the given lock is in fact held by the current process. The effect of these macros is to reliably document in the source code which locks are held where. In scanning the sources, either a lock grab or a lock assert should be present in every subroutine that attempts to update a locked data structure.

This technique also ensures that any new code can be easily tested for conformity to the existing rules on locked procedure calls.

Deadlock avoidance

There are two main forms of deadlock. The simplest method is to grab a lock and then call a procedure which calls a procedure which calls a procedure that again wants to grab that lock. Unfortunately this is a surprisingly simple crime to commit, particularly when using procedure variables, as is common in the socket code and in the generic file systems code. Fortunately it's a relatively easy crime to catch - it will show up the first time that code path is executed.

The harder method for deadlocking is to grab lock A and then to try to grab lock B at the same time as some other process has grabbed lock B and is trying to grab A. Each process then has a lock and won't release it until the other releases his. This method relies more on luck and on clever timing than on simple mechanistic stupidity. This makes it a hard crime

to detect - the two players might successfully grab their locks thousands of times before they run into a conflict.

Both of these crimes are complicated by interrupt handlers. A process might be quite happily plodding along occasionally grabbing and releasing lock A when suddenly it takes an interrupt which wants to grab lock B. This suddenly brings it into conflict with some other process which has happily been in the custom of grabbing B and then A.

This problem is partly a consequence of our design. If we were to adopt a strategy of only having a single lock for the whole kernel, or at least only having locks on major subsystems (such as "the filesystem" or "the network code") then maybe we could avoid ever having to hold more than one lock at a time. And this would avoid all the problems of lock ordering. But our strategy of having many locks means that we will often have to keep one resource locked while obtaining locks on second or third resources. E.g. we might want to keep a virtual memory segment locked, even while locking the free chain to obtain a page.

So there are lots of potential deadlocks which will only show up when a pair of processors are each in a particular state and one of them takes a particular interrupt. Normal beta testing will be unlikely to find all of these, but they will keep cropping up unless we have a strategy to avoid them.

So how did we avoid this? In two parts. First we adopted a strategy of ordering the locks. Then we built a system to enforce the rules.

Lock ordering

The main reason why processes deadlock is that they are in some sense moving in different directions through the code. One process is calling from module A into module B and the other is calling from B into A.

The standard kernel is already moderately hierarchical, with system calls representing the highest level and device drivers the lowest level. However, this hierarchy is fairly informal and is frequently broken. For example a "low level" disk device driver may call into "high level" buffer management code in the file system.

For our SMP prototype we aimed to construct a "hierarchical" lock order which corresponded approximately to the kernel's normal code hierarchy. Thus a process is only permitted to obtain locks in the order defined in the lock hierarchy. If he holds a "high level" lock he is permitted to claim a lower-level lock, but not the other way around.

Notice that a lock hierarchy is somewhat more flexible than a code hierarchy. Processes can call upwards through the normal code hierarchy, provided they are careful to release their low level lock before calling upwards.

We still have a problem whenever we have to

claim two objects of the same type which happen to have different locks. For example, when transferring virtual memory between the parent and child of a

Hierarchy rungs	Examples
High level file-system	file descriptors inodes namei cache
High level memory	segments page tables swap maps
High level network	sockets protocol chains
Devices	drivers teletypes busses
Low level file system	buffer pool
Low level network	interfaces ARP tables mbuf pool
Low level memory	page pool kernel alloc
Asynchronous events	select signals
CPU scheduler	run queues sleep queues cpu usage stats
Slime	hardclock timeouts console panic

Table 1

vfork, we need to lock the virtual memory data structures of both processes. The lock hierarchy cannot help us here and we need to adopt ad-hoc orderings for objects of the same type. E.g. we try to lock parent-then-child for locking related processes.

Interrupts complicate things slightly. For any given lock X, if X can be claimed by an interrupt handler at IPL P then everyone who wants to grab X must move to IPL P for the whole time they hold the lock. Otherwise, the interrupt may occur and the interrupt handler will try to grab the lock X which its processor already hold. Another form of self-deadlock.

It also follows that any lock Y of greater rank

than X must also only be claimed at IPL P. Otherwise, while holding lock Y we may take an interrupt that wants to claim X and then, following the hierarchy, tries to claim Y.

What this means is that we need to have an interrupt priority level associated with each lock, which also rises steadily with hierarchical rank ordering.

The main rungs of the lock hierarchy

We found it convenient to give every different lock type its own hierarchy number and ended up with 64 distinct ranks. However there are ten main rungs in the ladder, as shown in Table 1.

Notice that it prove convenient to divide the file system, the virtual memory system and the network code into distinct "high-level" and "low level" chunks. This generally represents a split between data structures which are manipulated exclusively from system calls and data structures which are also manipulated from interrupt handlers. However it also represents a real split between different levels of functionality - the high level virtual memory system relies on both the low level file system (for local swapping) and the high level network code (for remote swapping). But these two layers in turn depend on the low level virtual memory code for memory allocation.

We were a little surprised to find that device drivers ranked so high in the hierarchy. This occurs because so many low level structures (the buffer pool, the mbuf free list) can in fact be manipulated from device drivers.

Validating the lock hierarchy

So if the lock hierarchy and its associated interrupt priority level hierarchy are strictly followed then we can feel confident that we have avoided most forms of deadlock. But by itself, that doesn't really buy us very much. The kernel is a devious place. How can we actually convince ourselves that our hierarchy is valid and that it is universally obeyed? Time for a little consistency checking.

In debug mode we store a rank number and a lock IPL value in each spinlock object as part of its initialization. For each processor we keep track of which spinlocks it currently holds.

Whenever we grab a lock we check that its rank is greater than or equal to the ranks of any other locks we hold and we check that our current IPL is greater than or equal to the IPL of the lock. If either check fails, we panic, otherwise we add the lock to the list of those we hold.

Similarly, when we release a spinlock, we again check that its IPL is adequate.

Between them these simple checks validate that our lock strategy is being consistently followed. They are by no means infallible. If certain code paths for a single processor are not exercised during debugging, then these code paths might contain illegal lock ordering sequences. However, the checks do guarantee that

all the code exercised on a single processor successfully follows the rules and thus should avoid deadlock when faced with any combination of processors and interrupt handlers. This is much simpler than trying to test all these interactions individually.

Problems with the lock hierarchy

Unfortunately, a simple-minded hierarchy doesn't entirely reflect how the kernel really works. Not all calls are top down. Some important "upcalls" originate in device drivers and end up manipulating relatively high level structures. There are two important cases, the biodone mechanism and the network input code.

Block device drivers are required to call "biodone" when a buffer i/o operations completes. Normally biodone only updates the buffer pool, which is a valid operation in terms of the lock hierarchy. However, if B_CALL is set, biodone instead calls a procedure variable associated with the buffer. Fortunately all the existing uses of this mechanism already fit into the hierarchy, and we require any future uses to do so also.

The socket code is more awkward. Normal device input is initiated by a specific process which ushers it to completion. Socket input may be initiated by the arrival of an input packet at a network interface, after which the packet is progressively more closely identified and refined as it works its way through several protocol layers until it is finally delivered to a particular socket. This problem is solved by minimizing the locking that occurs on sockets. Rather than having a lock for each protocol layer associated with a socket, we have a single spinlock for each socket, which locks all operations associated with the socket. Thus, after the input socket has been identified, only one locking layer is used, even though the call progresses upwards through several logically distinct layers of software.

However, we are still faced with the need to move from a low level lock (the protocol control block chain) to a higher level lock (a specific socket lock). This poses an awkward problem. At other times we need to lock the protocol control chain when we already have a socket locked, so merely rearranging the lock hierarchy won't help. If we break our hierarchy, we shall surely deadlock, as there really are calls trying to grab the locks in different orders. We solved this class of problems by introducing a new lock operation which only tries once to grab the lock and then reports failure if it can't claim the lock. If this happens, the network code abandons the current action, releases the protocol chain lock and starts again from scratch. In the worst case, it may need to go around this extended spinlock loop several times before it succeeds in claiming both locks. Fortunately, conflicts on the socket lock at this level are extremely infrequent.

There are only a very small number of cases where we have found it necessary to use this "try

once" lock operation.

Numbers

Source numbers

In modifying the kernel for SMP we made the changes described in Table 2. This covers the core functionality of Ultrix 2.0 including GFS, NFS and the Arpa protocols, but excludes most other protocol

spin-locked data structures	60
mutex-locked data structures	10
calls on spinlock SGRAB macro	650
calls on mutex MUGRAB macro	70
calls on atomic free-and-sleep macros	100
calls on assertion macros	450

Table 2

families and most device drivers.

Performance for simultaneous makes

Selecting a fair and accurate benchmark for multiprocessor work is difficult. A CPU intensive benchmark such as a parallel ray tracer can be expected to show linear speedup with available CPU power, but this is neither interesting nor helpful.

We provide numbers for running a set of 8 large independent makes in parallel. This exercised most of the kernel and included over 1000 fork and execs, approximately 30000 i/o operations and about 80000 page faults. On a single 8300 CPU it runs for about 80 minutes of which approximately 15% is spent in the kernel.

Due to bus contention our modified 8300 does not provide linear CPU speedup for each additional CPU. Based on user CPU times, for our particular benchmark the second CPU adds about 87% of the first CPU and the third CPU about another 59%. To mask out this hardware effect, Table 3 gives performance in estimated numbers of instructions relative to the total instructions for the uniprocessor bench-

	total	user	sys	idle	speedup
One CPU	100	86	13	1	1.00
Two CPUs	103	86	13	4	1.95
Three CPUs	106	86	13	8	2.81

Table 3

mark.

Note that while our overall speedup is slightly less than linear, the amount of time spent in the kernel is almost exactly constant and the wasted time is entirely spent in the idle loop. The implication is obvious - as we add more CPU power our "realistic" benchmark becomes more i/o bound. This isn't totally surprising, particularly when we note that

even on a single CPU the benchmark is unable to exploit all 100% of the available CPU. C'est la vie.

The benchmarks were all run with a single disk controller. If we had added disk controllers and disks at the same time as we added CPUs, we might have seen a slightly more linear speedup.

Lock contention numbers

Assuming adequate hardware, the main factor determining how well this design will scale to a large number of processors is the degree of contention for kernel spinlocks.

Table 4 describes the spinlock contention rates for our "make" benchmark on three processors. In this case we also included a few TCP connections for logins and performance tools. "Average spins" is the average number of times we spin when the lock is busy. Our spin loop is approximately 50

Lock	grabs (1000s)	conflict rate	Average spins
process stats*	630	0.03 %	1.0
time-outs	460	0.38 %	1.1
time	230	0.01 %	1.0
fs buffer pool	290	0.94 %	3.5
free page list	220	1.48 %	1.5
cpu scheduler	180	1.37 %	1.4
vm segment*	130	0.04 %	3.0
copymap	100	0.62 %	2.3
TCP chains	70	0.04 %	2.4
biodone	70	0.07 %	2.7
disk*	60	2.61 %	13.5
bus map*	60	0.00 %	0.0
mbuf pool	40	0.13 %	4.4
gnode pool	36	0.26 %	1.9
interface queue*	29	0.00 %	0.0
kmemall	26	0.05 %	1.0
namei cache	25	0.17 %	1.3
swapmap	24	0.16 %	2.8
file table	19	0.00 %	0.0
socket*	17	0.66 %	8.2
file system struct*	14	0.11 %	5.0
proc table	14	0.00 %	0.0
credential pool	13	0.00 %	0.0

Table 4

microseconds.

Other lock classes which received between 1000 and 10000 accesses in this particular example included signals, IP protocol chains, select, teletypes, ARP tables and UDP protocol chains. (The network protocol code was only lightly exercised in this benchmark. Most of the TCP chains activity is due to timeout

* => locking occurs separately on each object of that class. Otherwise there is only a single lock for the whole class.

processing that is independent of load.)

These numbers show low contention rates for all locks, which should easily allow our design to scale beyond ten processors without encountering serious degradation due to lock contention. The longest time spent spinning (grabs x conflict rate x average spins) is for our single UDA disk controller's lock. Adding additional disk controllers with their own locks can be expected to reduce this contention.

The ultimate bottlenecks of our current kernel are likely to be the scheduling spinlock, the buffer pool, and the free list. All three of these locks control complex relationships involving multiple chains (e.g. the free list must be consistent with the text page hash chains); which will require significant restructuring and more complex locking in order to permit more concurrency.

In some ways these numbers may be misleading. They are the result both of final system tuning and also of early design decisions to try to minimize contention at notable potential bottlenecks. For example in both the file system and the virtual memory system we attempted to minimize the use of single global locks, preferring to use finer grain locks wherever possible. Additionally we performed some tuning to reduce the length of time that the locks on the page pool and the buffer pool were held.

A few other areas benefited considerably from tuning. Originally our scheduling spinlock was also used to lock the per-process CPU statistics, which are updated on each clock tick on each CPU. Introducing a separate lock class for this purpose, removing scheduling actions from each clock tick and tuning schedcpu and roundrobin considerably reduced the contention for this lock.

Similarly, we originally had one lock that controlled both the time of day and the callout chains. This lock was required on each tick on each processor. We introduced a separate lock for the time-of-day updates and optimized our timeout handling so that normally only one CPU needed to lock the timeout queues on each tick.

Conclusion

It was reasonably straightforward to add fairly fine-grain spinlocks to the Ultrix kernel and this appears to have resulted in acceptable kernel concurrency with only a limited degree of tuning.

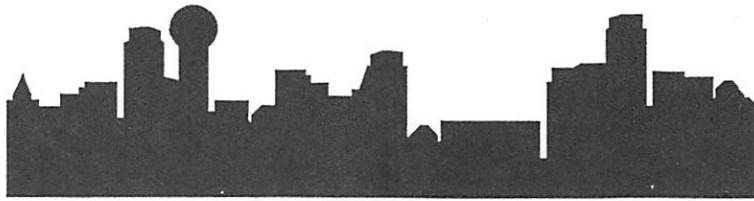
The use of a hierarchical locking system, of assertions and of considerable run-time consistency checking was extremely valuable in debugging the kernel. We also believe that it makes the kernel easier to maintain and to modify.

References

- [Bach84] M. J. Bach & S. J. Buroff, *Multiprocessor Unix Operating Systems*, AT&T Bell Laboratories Technical Journal, Vol. 63 No. 8,

October, 1984.

- [Beck84] R. D. Beck & R. A. Kasten, *Multiprocessing with Unix*, Systems and Software, October 1984.
- [Finger85] E. J. Finger, M. M. Krueger & A. Nugent, *A Multiple CPU version of the Unix kernel*, Usenix Conference Proceedings, January 1985.
- [Goble81] G. H. Goble, *A Dual Processor VAX 11/780*, Usenix Conference Proceedings, September 1981.
- [Gould85] E. Gould *Device drivers in a Multiprocessor Environment* Usenix Conference Proceedings, June 1985.
- [Inman85] J. Inman, *Implementing Loosely-Coupled Functions on a Tightly Coupled Engine*. Usenix Conference Proceedings, June 1985.
- [Probert86] D. Probert, J. Berkowitz & M. Lucovsky, *A Straightforward Implementation of 4.2BSD on a High-performance Multiprocessor*, Usenix Conference Proceedings, January 1986.
- [Test86] J. A. Test, *Concentrix - A Unix for the Alliant Multiprocessor*, Usenix Conference Proceedings, January 1986.



Joseph R. Eykholt
Amdahl Corporation
1250 East Arques Avenue (M/S 316)
P.O. Box 3470
Sunnyvale, CA 94088-3470

A New Exception Handling Mechanism for the UNIX Kernel

ABSTRACT

This paper describes an exception handling mechanism that can be used by the kernel to handle error conditions that occur during kernel processing. Error conditions may result from hardware faults, use of uninstalled features, addressing errors, or software detected errors. This exception handling mechanism may also be useful in application programs.

Introduction

In an effort to improve recovery from hardware faults, we developed an exception handling mechanism to allow C routines to specify where to resume execution after the hardware fault interrupt processing is complete. This exception handling mechanism can also be used for recovery from hardware detected software errors, such as the use of an uninstalled feature or an addressing error, or for recovery from software errors or signals.

When a hardware fault occurs, the fault interrupt handler may not have enough information to complete the recovery by itself. The recovery required often depends on what operations were being attempted; with some hardware faults, the instruction address at the time of the error may not even be known.

The *iostart()* routine, which starts I/O operations for device drivers, is an example of a routine that can take advantage of error recovery. If an error occurs while executing the hardware instruction to start the I/O, *iostart()* can retry the operation or simply return an error to the caller. This would confine the damage to one device or set of devices. The essence of this routine is shown below, using *save()* to establish a simple error handler:

```
iostart(device, operation)
{
    if (save(u_esav)) {
        /* error occurred */
        return(-1);
    }
    a_ssch(device, operation);
    /* start subchannel */
    return(0);
}
```

Save and Resume

The routines *save()* and *resume()* currently are the basic mechanism for handling signals that are posted while a process is sleeping. A review of these routines might help the reader understand the proposed new exception handling mechanism. The current uses of *save()* and *resume()* are to save the registers before switching to another process (using the save area *u_rsav*, and to declare the point where execution resumes if a signal occurs while the process is sleeping in a system call (using *u_qsav*).

Save() is the same as the C library routine *setjmp()*, and *resume()* is similar to the *longjmp()* library routine, except that *resume()* takes only one argument.

Save() simply saves the registers in the save area provided and returns zero. When *resume()* is called, it reloads the registers from the save area and returns a non-zero value. Since the registers have been reloaded, it appears to the caller that *resume()* never returns and that *save()* returns again from the point where it was called, this time with a non-zero value. Of course, all variables, except for those in registers, will have the value they had when *resume()* was called.

Since the registers saved and restored include the stack pointer, *resume()* will give unpredictable results if the routine that called *save()* has returned.

Error handler

An exception handling mechanism could use *save()* and *resume()*, with a save area in the user structure. When an exception occurs, processing would resume at the saved point. This is the method used in the *iostart()* routine shown above.

In that example, the protected code in *a_ssch()* could cause an error and the error interrupt handler would make it appear as if a *resume()* had been performed. The recovery action would take place and

return an error from the routine.

This simple technique has a problem, however. If an error takes place after *iostart()* returns, the error interrupt handler cannot resume the process from the state saved by the call to *save()*, because the stack frame for *iostart()* is no longer valid.

Exception handler stack

To eliminate this problem, a stack of exception handlers can be used. The choice of a stack is also motivated by the desire to save the environment that was established by higher level routines when a routine establishes its exception handling. The routine *err_save()* performs a *save()* and pushes the register save area onto the exception handler stack. Before returning, *iostart()* would call *err_pop()* to pop the exception handler stack. When an error occurs, the topmost save area is popped from the exception handling stack, and a *resume()* operation performed.

Storage for the stack could be provided using a fixed array in the *user* structure, but it is easier and more flexible to use a linked list of save areas allocated as local variables on the stack. A pointer to this structure, *u_err_save*, is added to the *user* structure. The save area structure, *err_label*, is declared below:

```
struct err_label {
    struct err_label *err_prev;
    label_t label;
};
```

Before the error handler is resumed, its *err_label* structure is popped from the stack. If an error occurs after that, the previously declared error handler, which will be at the top of the stack, will be invoked.

Organizing the exception handling stack as a linked list also has the advantage that the save area can be completely initialized before it is linked onto the top of the stack. The *err_save()* routine can perform the save operation; set *err_prev* to the previous exception stack pointer; and then in a single instruction, set *u_err_save* to point to the new save area. Any exceptions that occur before that instruction will be handled by the previously defined exception handler; those that occur after that instruction will be reflected to the new exception handler. With this mechanism, there is no danger of an exception occurring with a partially established save area on the stack.

The example becomes:

```
iostart(device, operation)
{
    struct err_label ps;

    if (err_save(&ps)) {
        /*
         * recovery action takes place here
         */
        return(-1);
    }
```

```
    a_ssch(device, operation);
    /* start subchannel */
    err_pop(&ps);
    return(0);
}
```

The routines *err_save()*, *err_pop()*, and another routine, called *err_resume()*, are defined below. Note that *err_save()* must be implemented in assembler or as a preprocessor macro, because the embedded call to *save()* must appear to be in the routine that called *err_save()*. *Err_pop()* can also be implemented as a macro.

```
err_save(esp)
struct err_label *esp;
{
    int rc;

    if ((rc = save(esp->label)) == 0) {
        esp->err_prev = u.u_err_save;
        u.u_err_save = esp;
    }
    return(rc);
}
```

```
err_pop(esp) .
struct err_label *esp;
{
    u.u_err_save = esp->err_prev;
}
```

```
err_resume()
{
    struct err_label *esp;

    esp = u.u_err_save;
    u.u_err_save = esp->err_prev;
    resume(esp->label);
}
```

Exception Trap Handling

When any interrupt occurs, the current state is saved on the process' kernel stack and the *trap()* routine is invoked to process the interrupt.

Because the interrupt may be unrelated to the interrupted process, exceptions that occur inside the interrupt handler should not be processed by the exception handling stack that was established by the interrupted process. The *trap()* routine saves the value of *u_err_save* in a local variable, and sets it to NULL. This establishes an empty exception handling stack for errors that occur in the interrupt handler. When trap processing is complete, *u_err_save* is restored from the local variable.

When an interrupt occurs that is related to the interrupted process because of an exception condition that should be handled by the exception handling stack of the interrupted process, the saved state on the stack is adjusted so that when interrupt

processing is complete, execution resumes as if *err_resume()* were called on the saved exception handling stack. The routine *trap_resume()*, which simulates *err_resume*, will be shown in detail later.

This method is preferable to calling *err_resume()* from the interrupt handler because it allows the normal flow of control in returning from the interrupt handler.

Refinements

Instead of making the example routine *iostart()* simply return an error, it may be desirable to retry the recovery several times; and if the error persists, invoke the previous error routine. Since an error interrupt will simulate a call to *err_resume()*, which pops the *err_label* from the stack, any error that occurs inside the error recovery code will automatically invoke the previous error routine. The recovery routine now looks like this:

```
iostart(device, operation)
{
    struct err_label ps;
    int errcnt;

    errcnt = 0;
    while (err_save(&ps)) {
        if (errcnt++ > 10) {
            err_resume();
        }
        /*
         * Recovery action takes place here
         * then loop back to repeat err_save call.
         */
    }
    a_ssch(device, operation);
    /* start subchannel */
    err_pop(&ps);
}
```

If the recovery action succeeds in the first ten tries, *err_save()* is reinvoked, and the protected code reattempts whatever operation caused the error. If the error occurs on tenth try, *err_resume()* is called to pass the error on to the previously declared error handler.

Special care must be taken when using local variables in the recovery code. The local variable *errcnt* could be assigned to a register by the compiler, and register variables are restored to their saved values when execution is resumed in the error handler. However, this example behaves correctly, since *errcnt* is not changed between the call to *err_save()* and the point where the error interrupt handler performs the resume.

Additional Refinements

The recovery code usually must know what error occurred, and this is given by the return value from *err_save()*. The task specific recovery code is cleaner

if the types of errors to be handled are specified in the call to *err_save()*, so that the caller can simply check for a non-zero return value. The *err_resume()* routine must find the most recently declared error handler for the error that occurred, pop it from the stack, and resume from its save area.

This refinement is shown by the following example, in which *errsave()* can be used to discover whether an instruction is installed by catching the error that is generated by the use of uninstalled instructions. Any other exceptions that occur will be caught by higher level exception handlers. The code to do this is as follows:

```
try_ipte()
{
    struct err_label err_label;

    if (err_save(&err_label, ERR_ILL) == 0) {
        a_ipte(test_pt, test_pa);
        /* try IPTE instruction */
        ipte_installed = 1;
        /* instruction worked */
        err_pop(&err_label);
        /* pop error handler */
    }
}
```

The above code fragment includes a test to see if the IPTE instruction is installed, and calls *err_save()* to establish an error handler for the exception that will occur if IPTE is not installed. (IPTE is the System/370 instruction "Invalidate Page Table Entry"). *Err_save()* returns zero at first, but will return non-zero if the *ERR_ILL* error occurs later. Since *err_save()* returned zero, the *if* statement is satisfied and *a_ipte()*, an assembler routine that executes the IPTE instruction, is called. If an operation exception occurs because of this, the *err_label* structure is "automatically" popped, *err_save()* returns with value *ERR_ILL*, and the routine exits. Otherwise, the *ipte_installed* flag is set and the error handler is popped by calling *err_pop()*.

The *copyout()* routine provides another example of how this can be used. It copies data from the kernel's address space to a user's address space. If the address supplied by the user points to an invalid or protected area, a kernel program exception will occur and will be handled by the exception handler. The code is as follows:

```
copyout(sadd, uadd, length)
caddr_t sadd, uadd;
int length;
{
    struct err_label err_label;

    /*
     * Setup error recovery in case the
     * user data area is
     * protected or the address is bad.
     */
}
```

```

if (err_save(&err_label, ERR_USEGV
            | ERR_PROT
            | ERR_ADDR)) {
    return(-1);
}

/*
 * copy from kernel to user address space.
 */
a_mvcs(sadd, uadd, length);
err_pop(&err_label);
return(0);
}

```

The routine simply returns a value of -1 if one of the three anticipated errors occurs.

Traps and Error Handlers

When the kernel gets an interrupt (for example, when *copyout()* is given a bad user address), the interrupt handler saves the exception handling stack pointer for the interrupted routine in a local variable, and restarts the stack to handle its own errors. The saved stack pointer can be manipulated to simulate the effect of calling *err_resume()* for the interrupted routine. The routine that does this is called *trap_resume()*. The essence of the final versions of *err_save()* and *trap_resume()* are seen in Figure 1.

Note again that the code for *err_save()* below must actually be written in assembler or as an in-line

macro, since *save()* must appear to be called from the caller of *err_save()*:

```

int
err_save(esp, mask)
struct err_label *esp;
{
    int    rc;

    if ((rc = save(esp->label)) == 0) {
        esp->err_mask = mask;
        esp->err_prev = u.u_err_save;
        u.u_err_save = esp;
    }

    return(rc);
}

```

The following version of *trap_resume()* causes a process that has been interrupted to resume in its error handler when the interrupt returns. This is machine and calling sequence dependent, by necessity.

```

/*
 * Cause resume in the error handler for a
 * process that got a machine check,
 * program check, or other error.
 *
 * Called from interrupt handlers to simulate
 * the actions of err_resume for the
 * interrupted process. The registers of the
 * interrupted process are restored from the
 */

/*
 * Structure used by err_save, err_pop, err_resume, and trap_resume.
 */
struct err_label {
    struct err_label *err_prev;    /* previous error handler stack */
    long    err_mask;             /* errors to be caught */
    label_t label;                /* register save area */
};

/*
 * Error codes used by err_save, err_resume, and trap_resume.
 * Values must to be single bits in a word, since they are used as
 * a mask for errors handled.
 */
#define ERR_UNKNOWN 0x0001 /* err_resume called with zero argument */
#define ERR_MCK     0x0002 /* machine check */
#define ERR_STACK   0x0010 /* kernel stack overrun */
#define ERR_KSEGV   0x0020 /* bad kernel address */
#define ERR_USEGV   0x0040 /* bad user address */
#define ERR_PROT    0x0080 /* write to protected address */
#define ERR_ADDR    0x0100 /* use of illegal address */
#define ERR_SYSCALL 0x0200 /* kernel-generated system call */
#define ERR_ILL     0x0400 /* illegal instruction */
#define ERR_PGM     0x0800 /* program exception
                        (other than the above) */
#define ERR_ALL     0xffffffff /* mask to specify all errors */

```

Figure 1


```

* err_label area, the return value (err_code)
* is placed in GPR0, and the
* instruction address is set up to continue 2
* bytes past the saved R14 value.
*
* The pointer to the error stack pointer is
* passed so that the error stack pointer may
* be updated to point to the new stack.
*/
trap_resume(regs, err_stackp, err_code)
struct u_uregs *regs;
struct err_label **err_stackp;
int err_code;
{
    struct err_label *esp;
    struct copy {
        label_t label;
    };

    if (err_code == 0)
        err_code = ERR_UNKNOWN;
    /*
     * Pop back to the most recent
     * error handler for this type.
     */
    esp = *err_stackp;
    while (esp && !(err_code & esp->err_mask))
        esp = esp->err_prev;

    if (esp == NULL)
        fatal(err_code); /* panic */
    /*
     * Pop the error recovery stack to the one
     * before the selected save area.
     */
    *err_stackp = esp->err_prev;

    /*
     * Restore registers from save area.
     * Then set the return value err_code.
     * Set the PSW address to the saved
     * return address + 2 to skip arg count.
     */
    * (struct copy *) &regs->gpr[2] =
        * (struct copy *) esp->label;
    regs->gpr[0] = err_code;
    regs->psw.psw_addr = (regs->gpr[14] + 2)
        & ADDRMask;
}

```

Forgetting the Pop

A drawback of this design is the need to pop the stack using *err_pop()*, and the possibility that this might be forgotten by a programmer and not discovered until some rare event occurred. One way to attack this problem is to define a macro that surrounds the protected code and includes the calls to *err_save()* and *err_pop()*. The protected code would be passed as an argument to the macro, which would look something like this:

```
#define ERR_SAVE(errs, handler, code) { \
```

```

    struct err_label err_label; \
    \
    if (err_save(&err_label, (errs))) { \
        handler; \
    } \
    \
    code; \
    err_pop(&err_label); \
}

```

A new version of the *copyout()* example using this macro follows:

```

copyout(sadd, uadd, length)
caddr_t sadd, uadd;
int length;
{
    ERR_SAVE(ERR_USEGV | ERR_PROT | ERR_ADDR,
        return(-1),
        a_mvcs(sadd, uadd, length) );
    return(0);
}

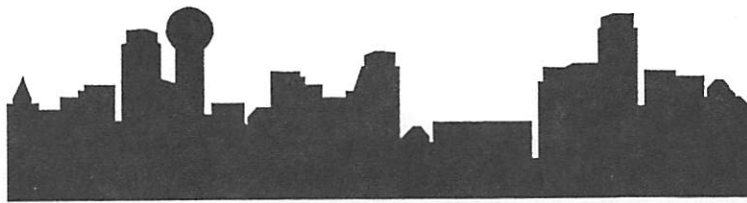
```

This is admittedly a bizarre use of preprocessor macros. Also, it doesn't prevent a goto or return from within the protected code. While I don't recommend the use of this strange macro, it did seem interesting enough to mention.

Another way to provide full protection against forgetting to call *err_pop()* would be to embed the exception handling mechanism into the language. However, doing that is undesirable, since it would also cause the *err_save()* mechanism to be unnecessarily "cast in concrete."

Summary

We have found this exception handling mechanism seems to be an elegant way of handling errors in our kernel. It would be just as useful for user-level programs, which would only require a global variable to be used in place of the user structure variable *u.u_err_save*. Perhaps many other applications for this mechanism will be discovered.



Translation Lookaside Buffer Synchronization in a Multiprocessor System

Michael Y. Thompson
J. M. Barton
T. A. Jermoluk
J. C. Wagner
c/o Silicon Graphics Computer Systems
2011 Stierlin Road
Mountain View, CA 94043-1321

ABSTRACT

Most current computer architectures use a high-speed cache to translate user virtual addresses into physical memory addresses. On machines that require software to implement cache fills and invalidations, the software task is fairly straightforward. In a multi-processor multi-cache configuration, however, where processes are allowed to migrate across processors, there is an inherent synchronization problem, as well as performance issues.

This paper discusses a solution to these issues that is general enough to implement without specialized hardware, yet offers good performance.

Introduction

Most current computer architectures use a high-speed cache to translate user virtual addresses into physical memory addresses (a *translation lookaside buffer*, or TLB). When a translation entry does not exist for a particular user virtual address, some combination of software and hardware must be employed to create that translation and supply it to the TLB. When a current virtual/physical translation changes or becomes invalid, as happens when a physical page is "stolen" from one process and assigned to another, an extant TLB entry must be replaced or removed. The methodology to perform these functions is well-known on a traditional single-processor (SP) computer system.

It was found, however, that the methodology available was insufficient when applied to a multi-processor (MP) configuration where processes are allowed to migrate across processors. In particular, the methodology fails on a multi-processor system where each processor is coupled with a private TLB: replacing or removing an entry in one TLB does not change or invalidate other, possibly extant, entries on other system processors.

This paper discusses the overall strategy that was devised to manage the TLB. The various situations in which TLB entries must be replaced or invalidated are enumerated, as are the details of both the SP and MP implementation.

Translation Lookaside Buffer

The target hardware is a system using the MIPS R2000 simplified-instruction-set processor. The TLB is part of the system coprocessor, one of which is associated with each processor. The TLB does not have a direct connection to memory, and it knows neither the form nor the location for page tables. TLB management is accomplished by software via coprocessor instructions. This approach requires slightly longer refill times than might occur with dedicated hardware, but has the advantages of simplified hardware and flexibility.[MIPS86]

Each TLB entry consists of two words. The low word contains a physical page frame number and various hardware bits (valid, dirty, etc.). The high word contains a virtual page number (VPN) and an id (TLBID). The id field is currently six bits — thus, 64 TLB ids are available. Additionally, there exist two index registers which are used to address TLB entries (an Index register and a Random register), and an EntryLo and EntryHi register pair. The formats of the EntryLo and EntryHi registers pairs are the same as the TLB entries. Figure 1 shows these formats. A TLB match occurs when there exists an entry which matches the input virtual address and the current TLBID field in the EntryHi register. Misses cause an exception, as do references to invalid entries, or stores to an address that matches a TLB entry that is not

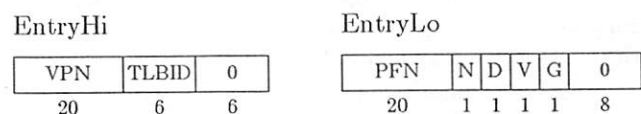


Figure 1. Translation Lookaside Buffer Format

marked dirty. Coprocessor instructions exist to probe for an extant entry (the index of the entry is left in the Index register); to read a specific TLB entry (EntryHi and EntryLo receive the contents of the TLB entry indexed by the Index register); to write to a specific entry (EntryHi and EntryLo via the Index register); and to write to a "random" TLB entry (the

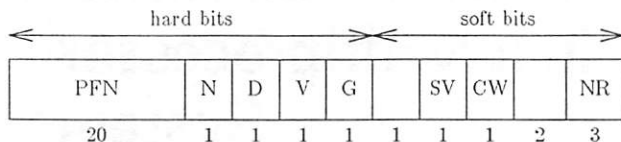


Figure 2. Software Page Table Format

pseudo-random Random register is used as the index). The page table is the software counterpart of the TLB. When a TLB entry is written, it is the software page table entry that is copied into the TLB. Bit fields not used by the TLB hardware are used for (software) valid and copy-on-write flags, and for a reference counter. Figure 2 shows the page table entry format.

Operating System Support

There are five different situations in which, on our system, (a port of the 5.3 UNIX Operating System) TLB entries can become inconsistent with process state. They are:

1. A process shrinking its address space.
2. Physical pages being "stolen" from a process.
3. System virtual address reallocation.
4. System physical address reallocation.
5. Writes to copy-on-write pages.

The first situation occurs when a process sets its maximum data value to a lower value, when it releases a shared memory segment, or when it releases all its address space on exit.

The second scenario occurs in low-memory situations when the memory management daemon takes physical pages from a process to be available to others.

The system also keeps a map of virtual addresses which are allocated for short durations for purposes such as mapping user physical addresses into system virtual space for DMA. After each use, the virtual addresses are returned to the system address map for reuse.

Similarly, physical pages are often assigned for varying durations to steadfast system virtual addresses such as file system buffers. Over time, pages may be assigned, usurped, and new pages assigned.

Lastly, when a process writes to a shared copy-on-write page, a copy of the page is created and the new page is assigned to the writing process.

In all of these situations, there can exist entries in the various TLBs that are suddenly incorrect. In

all of these situations it is necessary to ensure that the process doesn't access addresses that it has surrendered. To that end, there must be no entries in the TLB on the processor on which a user process is running that map virtual/physical addresses which are no longer correct. Similarly, the kernel process must take pains not to access kernel virtual addresses which are no longer valid.¹

In our original SP port, the TLB replacement and invalidation policies were situational. That is, for each situation an expedient method was devised to keep the TLB synchronized with the system state, but there existed no overall strategy for TLB management. On the MP system, it became clear that an over-all policy was essential, both to make the various mechanisms work efficiently (severally and together), and to make the problem manageable.

While on an SP system it was often appropriate to replace or remove TLB entries immediately as the entry became invalid, on an MP system this strategy suffers from overenthusiasm. It could well be the case that a process which has divested itself of pages or has had new physical pages assigned to particular virtual addresses never runs on other processors on the system, or runs on another processor only after "natural" events have caused the invalid entries to be replaced or removed. We decided to accentuate this tendency and put off TLB invalidations until absolutely necessary.

To implement this strategy (which we labeled "lazy devaluation"), system and process state is recorded to understand when TLB entries on a particular processor must be replaced or invalidated. When such an event does occur, the entire TLB is flushed, and the state structures are adjusted so that, logically, the flush creates the greatest effect.

The following sections explicate the various situations and mechanisms involved.

Shrinking Processes

There are several scenarios in which a process might divest itself of current address space. These range from a process resetting its break value to a process detaching a shared memory region or unmapping a mapped file region to a process exiting.

The last case is benign — an exiting process no longer has the ability to reference its address space.

The other cases are surmountable. Since a TLB match requires that an entry match both the input virtual address and the current TLBID (in EntryHi), assigning a new TLB id to the process effectively renders current (possibly stale) entries inaccessible. This approach is more efficient than the alternatives — flushing the entire TLB whenever a process shrinks its address space, or probing for and invalidating each possible (now invalid) TLB entry.

¹The implication is that, while user processes are not to be trusted, the kernel can certainly understand its own memory management state and take care not to abrogate its policies.

It is only when there are no readily available TLB ids that drastic action needs to be taken. In that case, each process' TLB id is set to an invalid value (the id is kept in the proc structure) and the TLB is flushed. It is safe to invalidate the id field of an active process since it is guaranteed that, on an SP system, no other process besides the one requesting an id is currently running, and thus, there is no process actively using TLB ids. When a process resumes, it checks if its TLB id is still valid; if not, it requests a valid id from the id allocator.

On an MP system, there is no such guarantee — processes on other processors may well be active. The TLB id reallocation problem is easily solved, however, by freeing only those ids whose associated process is not currently running. A field in the proc structure indicates whether the process is currently running on any processor. With suitable spin locks and semaphores to protect bit fields and TLB id allocation code, process shrinking becomes quite tenable for the operating system.

TLBIDs are managed as a site-wide resource, so, at the time that ids must be recycled all TLBs on the site must be flushed. To effect site-wide flushing, it is only necessary to set bits in a global bit field, one bit for each active processor. Whenever a processor flushes its TLB, it clears its corresponding bit in the field.. The initiating routine merely sets all appropriate bits beforehand, flushes its own TLB, and waits until the entire field has been cleared. (On systems that have an inter-processor interrupt facility, this wait is minimal. On systems without hardware support, simple messaging can be used to initiate TLB flushing on the various processors.)

Reclaiming Pages

The major functions of the paging daemon are to determine page usage and to free pages into the page pool when memory gets tight. As there are no hardware reference bits available, page usage on our system is determined by periodically decrementing software reference counters and turning off the hardware valid bits in the page table entries. The paging daemon has only to invalidate the corresponding entries in the TLB to cause subsequent references to produce reference faults. The fault code resets the valid bit and the reference counter for the faulting page, and drops the entry into the TLB.

Similarly, to reclaim a page for the free pool, the paging daemon clears the software and hardware valid bits in the page table entries, and inserts the pages into the free list. Semaphores associated with each virtual memory region [Bach86] are used to ensure that page faults and page manumission are, effectively, atomic.

For both reference fault enabling and page manumission, TLB entries are not invalidated individually. Instead, a number of pages, possibly spanning several regions, are operated on at once. Before the region semaphores are released, TLBs are flushed

site-wide.

System Virtual Addresses

In general, the operating system runs without TLB mappings. The kernel is divided into three segments which carve out the addresses from 0x80000000 through 0xffffffff (the user segment — kuseg — includes all virtual addresses from zero through 0x7fffffff). References to kseg0 (0x80000000 to 0xa0000000) are cached but not mapped into the TLB. Most of the kernel's executable code and some of its data reside here. The kseg1 segment (0xa0000000 to 0xc0000000) provides uncached, unmapped references — I/O registers and ROM code are mapped to these addresses. Both kseg0 and kseg1 addresses are direct-mapped onto the first 512MB of physical address space. Like kuseg, the kseg2 segment (0xc0000000 through 0xffffffff) uses TLB entries to map virtual address to arbitrary physical ones. The operating system allocates kseg2 addresses for some dynamic structures and for performing DMA into user

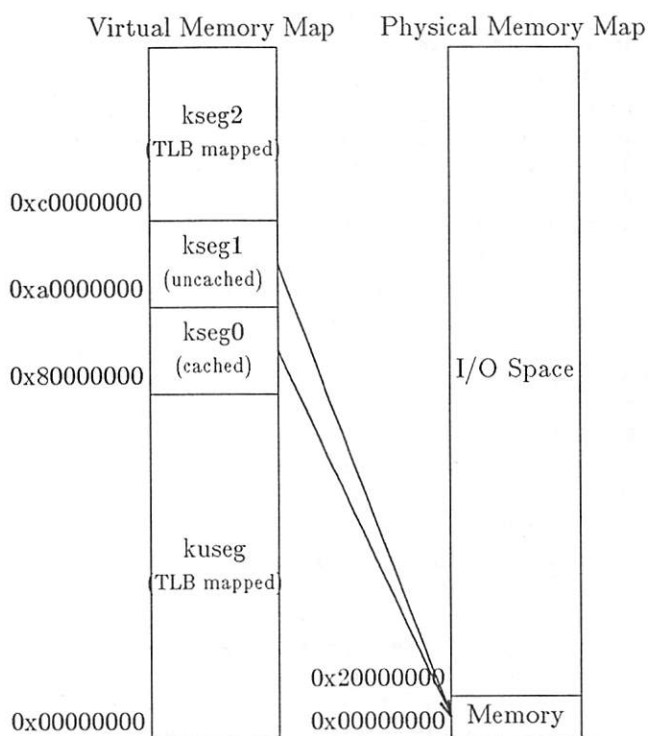


Figure 3. Hardware Defined Virtual Memory Map

space. For user DMA, the system allocates kernel virtual addresses from a system address map and double-maps the user's pages into the system space. The interrupt code which transfers data then does not need knowledge of the user process for which the transfer occurs. On an SP system, dropping in new TLB entries for the system virtual pages when they are allocated is sufficient to ensure that no stale TLB entries exist from the previous allocation. (The dropin code probes for a current entry for the TLBID/virtual-address pair and replaces that entry if

it exists.) But on an MP system, dropping in new TLB entries on one processor does not affect other processors' TLBs. Again, instead of signaling each processor and having each processor replace or invalidate entries, we take the lazy devaluation approach. The various TLBs are allowed to fill with new entries "naturally", that is, by reference. Upon deallocation, however, the page is not returned to the free map, but is instead placed in a stale address map. If the system map becomes depleted, the site-wide TLB flush routine is called. This routine always merges the stale address map back into the system map while waiting for other processors to flush their TLBs.

System Kseg2 Mappings

A variation of the page reclaiming problem exists with certain kernel routines that allocate and free physical pages associated with kernel virtual addresses (for example, pages for file system buffers). Unlike the memory management paging daemon, which frees large numbers of pages at a time, pages are released in small numbers. Because of this, wholesale TLB flushing is inappropriate. Instead, we apply the precept of lazy devaluation. We track page usage through state tables and postpone TLB flushing.

When a page is returned to the free list, the valid bits are reset, but the page frame number persists in the system page tables. It is only when the virtual address is surrendered that the page table entry's page frame number is cleared, indicating that there is no "remembered" association with a physical page.

When allocating a page for a system virtual address, if there exists a page frame number in the page table entry, the named page it is reassigned to the virtual address if the page is free. If the page is not available, the system-wide TLB flush routine is called. At this time, all invalid system page table entries have their physical page frame number fields cleared, indicating that there is no longer a residual relationship between the virtual addresses and the physical pages.

As a performance enhancement, when a physical page that was mapped to a system virtual address gets returned to the free memory list, its corresponding system page table entry is linked on a dirty list. The TLB flush routine traverses this list when clearing the physical page frame numbers. If a process is surrendering both the virtual and physical pages, this linking is not necessary — returning the virtual addresses into a stale map ensures that the system won't use the address without first flushing the TLB.

When a previously-assigned page is reassigned to the same virtual address, it must, of course, be dequeued from the dirty list.

Overloaded fields in the parallel disk block descriptor (DBD) [Bach86] are currently used for this chore. (The descriptors are unused since the corresponding pages are never swapped to disk, and, in the current implementation, the DBDs are not separably allocatable.) To facilitate dequeuing, a

doubly-linked list is used; in order to fit into the DBDs, the fields are actually offsets into the system page table.

System Page Table System Disk Block Descriptors

Page Frame #	SV	Forward	Back
(unused)	X		
(list head)	X	3	7
AAA	1		
BBB	0	6	1
0	0		
DDD	1		
EEE	0	7	3
FFF	0	1	6

Figure 4. System Page Table w/Stale Relationships

Figure 4 shows an example in which page table entries three, six and seven have been chained into the "stale relationships" list. Entry four is not chained — the virtual address was released with the physical page. Figure 5 shows the same system page table entries after a system-TLB flush.

System Page Table System Disk Block Descriptors

Page Frame #	SV	Forward	Back
(unused)	X		
(list head)	X	1	1
AAA	1		
0	0	0	0
0	0		
DDD	1		
0	0	0	0
0	0	0	0

Figure 5. System Page Table After TLB Flush

Faults — Misses, Reference, Protection

The strategy for handling TLB misses is fairly straightforward. For first-level misses, the page table entry is copied to EntryLo, the VPN/TLBID pair is written into EntryHi, and the pair is randomly deposited into the TLB. Second-level misses are handled in a similar manner, except that the second-level entry (the TLB entry for the page table itself) is deposited into a specific TLB location, that location determined by software. On the current implementation, the processor constrains the Random register to contain a value from eight to 63. This allows entries zero through seven to be reserved for page tables, the

kernel stack, and the like.

Reference faults and protection faults are handled similarly — the page table entry for the faulting address is fetched (possibly causing a second-level miss), sanity checking is performed, and the new entry is dropped in, either replacing an extant entry, or, if none exists, dropped into a random TLB location. When a valid reference is made to an address to which a physical page is not currently assigned, the fault code must assign a physical page for the process and fill it appropriately.

None of these actions are a problem on an SP system, and, for the most part, on an MP system. Dropping an unchanged entry into a TLB is innocuous. Dropping in an entry for a newly-assigned page is trouble-free, too — the assumption is made that the routine that disassociated the page from its previous process/address took care to purge the (possibly extant) entries from the TLB(s).

Protection faults pose a problem on an MP system, however, when the fault is on a copy-on-write page. A copy-on-write page might be referenced by multiple processes at the time one process writes to it. The SP approach is simple: if more than one process is currently referencing the page, a new page is assigned for the writer, the data are copied, and a new TLB entry is deposited (with the dirty bit set). But on an MP system, there could exist entries on other TLBs (had the process previously run on other processors) that reflect the previous virtual/physical mapping. If the process migrated to another processor (on which it had previously run) without ensuring that the entry was purged, further references could access the wrong page.

Again, the approach is to keep state tables and avoid action until necessary. Instead of actively searching out and removing entries on TLBs throughout the system, it is just noted that there exist (possibly) stale TLB entries for this process on other processors. The minimal data structure is a bit field the size of the number of processors in the system, one for each TLB id (or, one for each proc structure, for small systems). After assigning a new page, it is only necessary to set the bits corresponding to all but the current processor, indicating there might be invalid entries for this TLBID (process) on the flagged processors. (A new entry is deposited in the current processor's TLB, so it is not necessary to set the dirty bit for the current processor.) When a process resumes, it checks whether a bit is set for the processor on which the process is now running. If so, the current processor's TLB is flushed. To further performance, a parallel bit field is kept that indicates the processors on which the process has actually run. It is only necessary to set dirty bits for processors other than the current one on which the process has previously run.

When a process is assigned a new TLB id (either because the process is just starting up, or because it shrank, or because its id was taken away for

reassignment), dirty bits in the entry indexed by the TLB id are cleared, as are the history bits for all but the current processor. (TLB ids are delivered "clean", that is, without any entries in any TLB using that id.) Similarly, whenever a processor flushes its TLB, the dirty bits for that processor in all entries are cleared, as are the history bits in all entries except for the

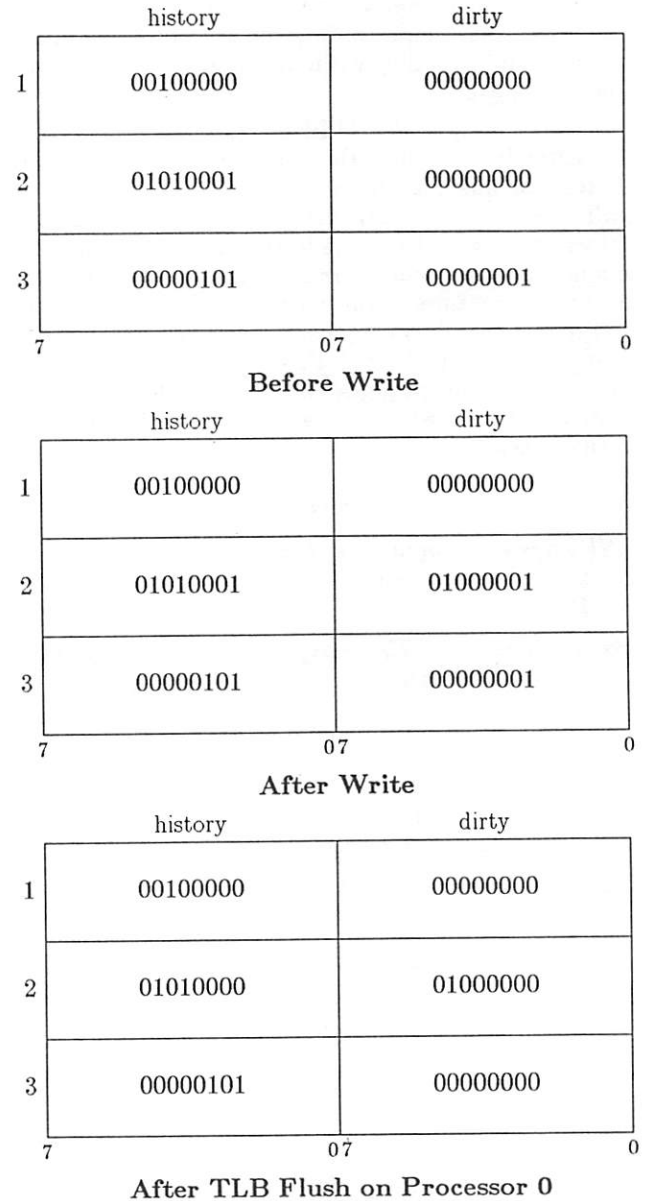


Figure 6. Example TLBID State Structures

currently running process. Figure 6 shows three snapshots of an abbreviated TLBPID state structure. The first and second are just before a process owning TLBPID 2 running on processor 4 writes to a shared copy-on-write page. At the time of the write, the process has a history of having run on processors 0, 4 and 6. The last snapshot is the same TLBPID state structure just after processor 0 has flushed its TLB. Note that the persistence of the history bit 0 for TLBPID 3 implies that it is currently running on processor 0.

Summary

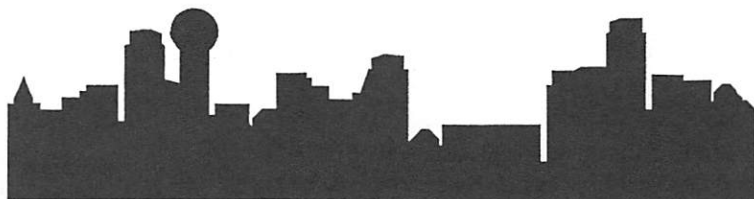
There are certainly other approaches that might have been followed to solve the problem of keeping multiple TLBs correct. Preliminary performance figures, however, indicate that the lazy devaluation approach succeeds without causing excessive TLB flushing.

Most importantly, the various state structures can be enhanced and routines tuned to take advantage of added information without changing the underlying mechanisms.

For example, the TLBPID state structures could be extended to list the individual stale entries. Instead of flushing the entire TLB, those entries (if still extant) could be individually flushed. If TLB id information were to be made available to the memory management daemon (currently, there is no path from a region structure, upon which the daemon operates, to process and TLB ids), a similar refinement could be made. It is not clear if these changes would be of benefit, but the changes could be implemented and tested without restructuring the entire TLB management system.

References

- MIPS System Programmer Guide, Beta Version*. Mips Computer Systems, Inc., Mountain View, CA, 1986.
- Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice Hall, New Jersey, 1986.



Adding Packet Radio to the Ultrix Kernel

Clifford Neuman¹
Wayne Yamamoto
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

ABSTRACT

This paper describes the results of a project in which we added the standard Amateur Packet Radio network link layer protocol, AX.25 (a modified version of X.25), to the Ultrix kernel. By implementing AX.25 under Ultrix, and by taking advantage of the IP implementations that already exist for PCs, we have made it possible for packet radio users with PCs to access IP-based services running on our server and on the Internet. We have dedicated a MicroVAX as an IP gateway for an Amateur Packet Radio network that stretches from Seattle to Tacoma.

Introduction

Packet Radio is an increasingly active area of experimentation among amateur radio operators. Stations consist of a radio transceiver connected to a terminal or a computer by means of a device known as a Terminal Node Controller (TNC). The TNC is essentially a modem. It "packetizes" data in a manner conforming to the AX.25 link layer protocol, provides a command interpreter, and has a primitive network layer protocol for use with terminals unable to support this layer on their own. Users with computers rather than terminals have the option of disabling the TNC's network layer protocol and providing their own.

Amateur Packet Radio has evolved somewhat chaotically. Initially, most packet radio stations consisted of terminals instead of computers. Once users had established communication with one another, they simply typed streams of data at each other. The TNC was functionally equivalent to a telephone modem.

One early development arose because users wanted to communicate with stations that couldn't be contacted directly. Relay stations were set up in strategic locations so that messages could be received and passed along to their destination. These relays are known as digipeaters. The standard amateur packet radio link layer protocol² allows the specification of

up to eight digipeaters through which a packet is to pass. This type of routing is known as source routing.

Another development was that some users connected their TNCs to computers on which they ran packet bulletin board software. This allowed others to access their computer in a manner similar to the way one accesses a BBS by phone using a modem. Users with terminals were able to leave messages and read messages. Other users who connected their TNCs to computers were able to upload and download files. The BBSs would forward mail to other BBSs for non-local users using packet radio.³

As the number of users of the amateur packet radio network increased, the demand for better connectivity between different parts of the country increased as well. As a result, network layer protocols became a topic of much discussion. A number of network layer protocols were proposed, and work has proceeded on many of them in parallel. The approach taken by COSI [1] and NET/ROM involved the establishment of networks of servers within which routing was taken care of automatically. With NET/ROM, users would connect to a node on the network. They would then connect to the NET/ROM node nearest their destination. Finally, they would connect to their destination. A similar approach is used for COSI, the main difference being that instead of actually connecting to a COSI node, one connects through it as if it were a digipeater. Users still had to know the name of their local node and the name of the node closest to

radio community. True network layer functionality is provided through other mechanisms, some of which are discussed in this paper.

³Usually one or two BBSs in each area would connect to station in different parts of the country (or the world) in order to forward messages from one packet network to another. In this way, connectivity for electronic mail was achieved on a world wide level.

¹This work took place while the first author was supported by NSF Grants DCR-8420945 and CCR-8619663 and the second author was employed by AT&T Bell Laboratories and attending the University of Washington. Computing equipment was provided by Digital Equipment Corporation's External Research Program.

²This is actually network layer functionality, but because packets are digipeated on the same frequency, and thus within the same subnet, digipeating is often considered a link layer function instead of a network layer function within the amateur packet

their destination. This was fine for users with terminals, but for users with their own computers, something better was needed.

At the same time that development was proceeding with NET/ROM and COSI, work was proceeding on supporting a packet radio implementation of TCP/IP for the IBM PC. This work was being done primarily by Karn [4]. One advantage of TCP/IP over the other approaches is that the user's computer becomes part of the network: one connects to the ultimate destination, rather than connecting to a network node and from there connecting to the destination. Another advantage is that users gain the ability to access IP services which are more varied than services accessible through the other approaches.

Although packet radio implementations of IP exist for several computers, the IP suite of protocols is not yet widely used in Amateur Packet Radio. One reason is that many users are still using terminals instead of PCs. These users can't run IP since they only have access to the limited software that runs in the TNC itself. There are also many users with computers for which there currently is no implementation of TCP/IP. For many of the users that do have computers capable of running IP, a major concern is that they will be isolating themselves from the users that can't run IP.

Another reason that the acceptance of TCP/IP is not greater is that much of the value of the Internet protocols is felt when one is using services that aren't available using other protocols or accessing systems that would not be accessible otherwise. Unfortunately, the existing implementation of TCP/IP for PCs only supports telnet (Remote login), SMTP (mail), and FTP (file transfer). These services are already available to users without using any of the higher layer protocols by connecting to a BBS and either reading or leaving mail there, or uploading and downloading files. Further, despite the fact that packet radio users of IP speak the same protocol as other systems on the Internet, and despite the fact that they have a block of addresses assigned to them by the Network Information Center, there did not exist a path by which they could connect to conventional Internet sites.

One of the primary objectives of our project was to provide a gateway between packet radio users (or at least, those that speak IP) and the Internet. This allows those users to access many of the network services that we, as Internet users, are used to. It is hoped that access to such services will stimulate the development of services specifically suited to the amateur packet radio community. The availability of such services will provide additional incentive for further stations to begin using IP. Another goal was to provide a gateway between users speaking other protocols over packet radio, and systems running IP. Such a gateway would allow stations to run IP without isolating themselves from the existing amateur packet radio network.

Implementation

We chose to achieve the goals outlined in the preceding section by adding support for packet radio to a system running Ultrix that was already on our department's Ethernet and part of the Internet. The code we used to encapsulate and decapsulate packets on our MicroVAX is based on the existing code for the PC.

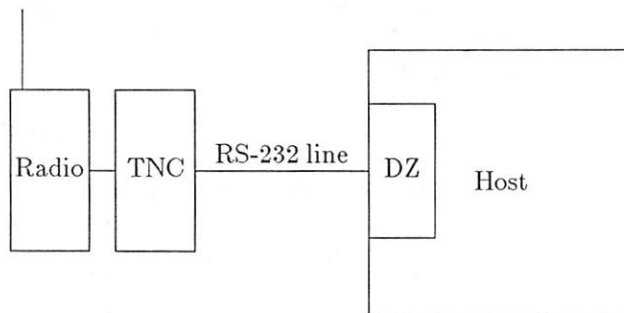


Figure 1. Physical hardware

The Hardware

As shown in figure 1, the special hardware used by our system to send packets by radio includes a radio and a TNC. The radio corresponds to an Ethernet transceiver, and the TNC to the Ethernet controller. One difference, though, is that the TNC does not sit on the bus. Instead, one communicates with it through a serial line. Since we did not require the higher software layers of the TNC, we used a stripped down version of the software for it known as the KISS⁴ [2] TNC code. All this code does is send and receive data and calculate the necessary checksums. Unlike the normal code that runs in the TNC and resides in the ROM of the TNC, the KISS TNC code, which may be downloaded into the TNC, does not worry about the packet format at all.

The Driver

In adding packet radio support to the Ultrix kernel, we implemented a pseudo-device driver for the packet radio controller. This driver supports the same calls as the drivers for other network devices such as the DEQNA. Since the packet controller does not sit on the bus, communication with it is through a serial line, and hence the driver is a pseudo-driver. Figure 2 shows the function of the hardware and software with respect to the ISO/OSI reference model.

In order to get the kernel to recognize the packet radio interface, we had to create and initialize a structure of the type `if_net`. The `if_net` structure contains pointers to the procedures used to initialize the interface, send packets, change parameters, and perform other operations. We had to create kernel procedures to perform each of these operations. The most difficult routine to write was one which handled incoming packets from the TNC. When a packet is received by the TNC, the TNC sends the packet as a

⁴The name is from the acronym "Keep It Simple, Stupid".

stream of bytes to the tty line. For each character in the packet, the tty driver calls the packet radio interrupt handler to process the character. Characters are buffered by the interrupt handler until all characters in the packet have been received.

As each character is read by the interrupt handler, some processing of characters is done on the fly. In particular, we decode escaped frame end characters that are embedded in the packet. When the final frame end is read, meaning that the entire packet has been received, the interrupt handler checks the header of the packet. It verifies that the recipient's amateur radio callsign (which is used as a link address) is either its own, or the broadcast address. It also checks the protocol ID field. If the packet type is IP, the driver then adds the encapsulated IP packet to the queue of incoming IP packets so that it can be dealt with by the existing Ultrix software. This approach to handling incoming packets allows other layer three protocols to be handled in an interesting manner, described later in this paper.

ISO/OSI Model	Protocol	Implementation
Application [7] Presentation [6] Session [5]	SMTp Telnet FTP	Existing Ultrix Network Support
Transport[4]	TCP or UDP	
Network[3]	IP	
Link[2]	AX.25	Packet Radio Driver
Physical [1]	Radio	TNC/KISS
		Radio

Figure 2. Comparison to ISO/OSI reference model

Setup and Testing

Once we had the packet radio driver running, the final task was to translate Internet addresses into AX.25 addresses. This is done using the address resolution protocol (ARP) [8] in a manner similar to the way that IP addresses are translated into Ethernet addresses. AX.25 addresses look like amateur radio callsigns followed by a 4 bit system ID. Things are complicated by the fact that some entries may contain additional callsigns for digipeaters. Thus, a different set of ARP routines is needed for packet radio. Karn's IBM-PC code [5] includes an ARP implementation that supports both AX.25 and Ethernet addresses. Because we did not want to modify the code for our system that is used on the Ethernet side of the gateway, we decided not to take this code. ARP lookup occurs at layer two, and thus, gets called inside either the Ethernet driver, or the AX.25 driver. The routing tables at the IP layer determine which driver is called. Since the ARP lookup occurs inside our code, we are able to call a separate routine that deals specifically with AX.25 addresses.

When the implementation was complete we enabled the packet radio interface at the Internet address of 44.24.0.28.⁵ We then modified the routing tables of another system on our Ethernet so it knew that 44.24.0.28 was the address of a gateway to net 44. After a few rounds of debugging, we were able to telnet from an isolated IBM PC⁶ to a system that was on our Ethernet by way of the new gateway. Since then we have used the gateway for file transfer, electronic mail, and remote login in both directions.

Future Work

In addition to providing a gateway between the packet radio network and the rest of the Internet, we would like our gateway to be able to serve as a gateway between applications running on top of other protocols. Such a gateway would be at the application layer, and specific to remote login and electronic mail. The way AX.25 was implemented in the kernel, such applications do not require kernel support, even though they extend down to layer three of the ISO reference model. Packets that are received from the TNC that are not of type IP can be placed on the input queue for the appropriate tty line. A user program can then read from this line, and maintain the state required to keep track of AX.25 level 3⁷ connections. Data can then be passed to a pseudo terminal to support remote login, and to a separate program to support electronic mail.

Work is also proceeding on using another layer three protocol known as NET/ROM to pass IP traffic between gateways. Doing this would allow the use of an existing, and growing, point-to-point backbone in

⁵Net 44 is assigned to Amateur Packet Radio by the Network Information Center.

⁶Connected to only a power outlet and a radio.

⁷AX.25 level 3 is the connection protocol supported inside the TNC.

the same way Internet subnets are connected via the ARPANET.

Performance

Because the link speed is only 1200 bits per second, the transmission time is the dominant factor in determining throughput and latency. Higher bandwidth links are available, but, at the moment, the hardware is not readily available at a reasonable price.

One performance problem that we noticed is that the gateway slows considerably as traffic on the packet radio subnet climbs. Part of the reason for this is that the present code running inside the TNC passes every packet it receives to the packet radio driver regardless of the destination address of the packet. We are considering changing the TNC code so that it can selectively pass only those packets destined for the broadcast or local AX.25 addresses.

Issues

The ability to interconnect amateur packet radio networks and non-radio networks introduces a few problems that have not been completely resolved as of this time. The three main problems are timeouts, routing and access control. In this section, we present these problems and suggest some possible solutions.

Timeouts

The problem with timeouts arises from the difference in the latency for the two networks. Hosts on the Ethernet side expect fast response. If they don't get a response quickly, they time out and retry their transmission. We have found that when connected to a system on our Ethernet from a machine on the packet radio side of the gateway, the system on the Ethernet side initially retransmits packets several times before a response makes it back. This results in wasted bandwidth as packets are needlessly retransmitted. Since these retransmissions are queued at the gateway, they delay other packets. Fortunately, many implementations of TCP dynamically adjust their timeout values. Hence, when the system on the Ethernet side learns the correct timeout value, the frequency of unnecessary packet retransmissions is reduced.

Internet routing

The problem with routing arises when we want to allow communication with Internet hosts beyond our Ethernet. In order for a response to come back, all the gateways between the source and the destination must know the route to the appropriate packet radio subnet. A class 'A' network is allocated for AMPRnet⁸, and since most systems by default will maintain a single route for a class 'A' network, all packets destined for AMPRnet originating from

another internet host, must pass through a single gateway. This is not desirable since a packet destined for 44.24.0.5 should be sent to a West Coast gateway and introduced to the packet radio network there, whereas a packet destined for 44.56.0.5 should be sent to an East Coast gateway. It is conceivable that something like this could be handled using ICMP⁹ redirects, but at this time, no mechanism is in place.

Access Control

Another problem we face is access control. Since operation is on frequencies assigned to the amateur radio service, any communication must be initiated by licensed amateurs. This is, in fact, an instance of a more general problem of trying to control access to a subnet so that hosts on the subnet can use services beyond the subnet, but, at the same time, hosts on the subnet are protected from adversaries beyond the gateway. One way to solve this problem is to maintain a table of authorized addresses on the non-amateur side of the gateway. Associated with each of these addresses is a list of hosts on the amateur side of the gateway with which that host can communicate. Initially the table starts off empty. Whenever a packet is received on the amateur side destined for a non-amateur host, an entry is made in the table, enabling the non-amateur host to send packets in the other direction. After a certain period of time, these entries are removed if packets have not been received from the amateur side of the gateway.

This scheme can be augmented with a few new ICMP messages. One message can force an entry to be removed from the table of authorized non-amateur systems. This allows the amateur radio operator that initiated the link to exercise his control operator function to cut off the link if he detects inappropriate use. Another message would allow one to add an authorized non-amateur host to the tables with an appropriately chosen time-to-live. Both these messages are allowed to come from either side of the gateway, but if they come from the non-amateur side, they must include a call sign and a password for an authorized control operator for the gateway.

Discussion

Packet radio is one of several options available when setting up computer networks. It is not the ideal solution in all situations. There are a number of problems that make non-radio communications more practical. Among the disadvantages are that radio communication is susceptible to eavesdropping, jamming, and impersonation¹⁰. Another disadvantage is that the radio spectrum is limited. Where wire or fiber-optic communications is an acceptable alternative it should be used.

Despite the disadvantages, there are situations where packet radio would be the technique of choice.

⁸AMPRnet (AMateur Packet Radio NETwork) refers to the subnetwork of the Internet consisting of amateur packet radio hosts, and with Internet addresses of the form 44.*.*.*.

⁹Internet Control Message Protocol.

¹⁰These can be solved using encryption and spread spectrum, but this increases the cost and might make licensing more difficult.

Mobile stations cannot be physically connected to one another. Packet radio is also useful for emergency field communications where one doesn't have the time to string wires. Another reason that packet radio is useful for emergency communications is that in a large scale emergency, such as an earthquake, land based communications will often be disrupted.

In a number of years, wide scale computer networking will be available to the public through services such as ISDN. Until that time, packet radio allows users, over a large geographic area, to set up networks connecting personal computers and larger computer systems. Use of a system running Ultrix as a gateway and server on a packet radio network is desirable since it allows users of PCs to access many of the services that one is used to having available in the Internet community. Telnet, FTP, and SMTP have all been successfully used across the gateway. We would like to support additional services, some of which are beyond those provided on the Internet. The Ultrix operating system can serve as a platform upon which these services will be built.

One useful service for the amateur radio community might be a distributed callbook. Many amateur radio operators purchase a book each year that contains the callsigns, names and addresses of every other amateur radio operator in the world. These books are organized by callsign, and are used to mail QSL cards.¹¹ With a distributed callbook server, data for a particular country, or part of a country, could be maintained on a system local to that area. Given a call sign, an application running on a PC could determine what area the call sign is from, and then send off a query to the appropriate server.

Allowing users to add information to their own entries (such as geographic coordinates) would allow many possibilities. It would be possible for someone establishing communication with anyone in the world to type in the station's call sign, and have their antennas automatically rotated to the correct bearing. Or perhaps, as a contact is made, one's computer can print out a mailing label for the QSL card. Someone even suggested having a bitmap of the QSL card sent to the other station through the packet radio network itself.

Distributed computation presents further possibilities. By giving the general public access to computer networks, the number of processors accessible on a network greatly increases. At the same time, the power of the processors will typically be less than those currently part of the Internet. This provides added incentive for users to come up with ways to distribute work to machines that aren't being used to their capacity.

In designing and implementing distributed services, it is important to note that there is a difference between the environment inside a university or

corporation with central control, and the environment of a network of PCs, where little central control exists. A network such as the one described in this paper can serve as an interesting testbed for applications that must run in such a decentralized environment.

In order for packet radio to reach its full potential for connecting otherwise isolated computers, it would be helpful for a few frequencies to be allocated outside of the amateur radio service. When using packet radio on amateur frequencies, there are lots of restrictions on the type of data that can be passed that makes productive use less practical. If there were a few non-amateur frequencies set aside for packet radio, and if the requirements and restrictions for operation on these frequencies were less stringent, one might see even more done with packet radio.

Conclusions

The Ultrix operating system provides a nice base upon which network services can be provided for the amateur packet radio community. At the same time, such a system can serve as a central node in the interconnection of local area networks running IP, and even those that don't run IP. By linking packet radio networks with more established networks, additional services become available. Such services are available in the Seattle area. These services are necessary if we are to interest people in running TCP/IP. Further, interconnection with non-IP packet radio users is necessary if we are to interest users who would like to try IP, but still want to maintain connectivity with those using other protocols.

Acknowledgments

A number of people were helpful in getting our implementation running and in discussing some of the ideas presented in this paper. Among them are Bob Albrightson (N7AKR), Mike Chepponis (K3MC), Bob Donnell (KD7NM), Dennis Goodwin (KB7DZ), Robert Henry (WA6BEV), Ron Hoffmann (WA2EYC), Ed Lazowska (KG7K), and Steve Ward (W1GOH).

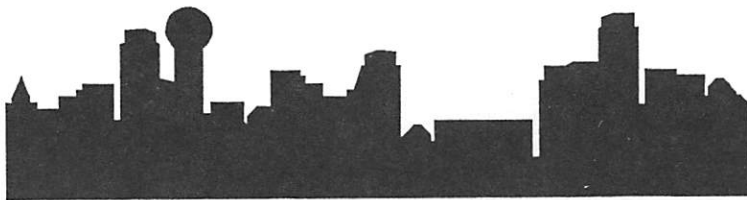
Bibliography

1. Beattie, J. Gordon Jr., and Moulton, Thomas. "OSI: A Plan Comes Together". Sixth ARRL Computer Networking Conference, Redondo Beach, CA, August 1987.
2. Chepponis, Mike, and Karn, Phil. "The KISS TNC: A Simple Host-to-TNC Communications Protocol". Sixth ARRL Computer Networking Conference, Redondo Beach, CA, August 1987.
3. Fox, Terry L. "AX.25 Amateur Packet-Radio Link-Layer Protocol. Version 2.0." American Radio Relay league, October 1984.
4. Karn, Phil. "TCP/IP, A Proposal for Amateur Packet Radio Levels 3 and 4". Fourth ARRL Computer Networking Conference, San Francisco,

¹¹A QSL card is a card confirming that a contact took place. Amateur radio operators often collect these.

March 1985.

5. Karn, Phil. "The KA9Q Internet (TCP/IP) Package: A Progress Report". Sixth ARRL Computer Networking Conference, Redondo Beach, CA, August 1987.
6. Leffler, S., Joy, W., Fabry, R., and Karels, M. "Networking Implementation Notes 4.3 BSD Edition". Computer Systems Research Group, University of California, Berkeley, June 1986.
7. Neuman, Clifford. "Packet Radio and IP for the Unix Operating System". Sixth ARRL Computer Networking Conference, Redondo Beach, CA, August 1987.
8. Plummer, David C. "An Ethernet Address Resolution Protocol". Network Information Center, RFC826. September 1982.



Man-Machine Interfaces for software development environments (HandS)

Eiji Kuwana
NTT Software Laboratories
1-9-1 Kohnan Minato-ku, Tokyo, 108
JAPAN

Hironobu Nagano
NTT Software Engineering Center
1-9-1 Kohnan Minato-ku, Tokyo, 108
JAPAN

Yuzou Nakamura
NTT Software Laboratories
1-9-1 Kohnan Minato-ku, Tokyo, 108
JAPAN

ABSTRACT

Man-Machine Interface (MMI) design and implementation for large scale software development environments utilizing Workstation (WS) and personal computer (PC) MMIs are presented. In this environment, WSs or PCs are used as front-end systems of mainframe computers. This is called a vertical software development environment. A prototype MMI vertical software development environment named HandS is presented. The HandS system can easily be used as an input/output front-end MMI system for various software development tools. HandS operates on UNIX systems with high resolution bit-mapped terminals and PCs.

Introduction

Sophisticated MMIs such as multi-windows, mouse, and menu-driven interfaces are most prevalent in end user environments [1, 2, 3]. Compared to those in workstation environments, MMIs in large scale Software Development Environments (SDEs) are still not efficient. For example, only screen oriented editors can currently be used. To provide large scale SDEs with adequate MMIs, an MMI system which is a systematically combined environment of workstations and mainframe computers is investigated.

In this research, UNIX workstations with bit-mapped terminals and PCs are used as the front-end system of the SDE for mainframe computers. This paper discusses the MMI design for the SDE, and then describes the prototyped MMI system (**HandS**). The HandS system supports many important features, including the HandS Scenario (H-Scenario) mechanism, a full screen editor, a window system, and cooperation between bit-mapped terminals and the UNIX environment by RS-232C or RS-422A.

HandS Scenario is aimed at automating regular software development processes, accumulating and re-using software development processes and know-how, and parallel execution of software development processes. H-Scenario is slightly like traditional

command procedures in its exterior syntax. Traditional command procedure systems, for instance, UNIX csh, are used for user command customization. With H-Scenario, however, it is possible to write MMI operations, such as window, menu-driven, iconic interfaces, and interactive screen editor control functions as software development commands. By providing interactive editor function control, H-Scenario can easily be used as an input/output front-end interface system for various software development tools. In HandS environments, end users can develop software efficiently by using H-Scenarios with rich and powerful MMIs. Therefore, HandS can greatly reduce programming development efforts because of improved MMIs.

In section 2, we present the MMI basic design concept for SDE based on software development process analysis. In section 3, we briefly describe the HandS system characteristics, and in section 4, we show the HandS configuration. Finally, in section 5, we present several H-Scenario examples for software development task assistance.

Software development process and MMI

In this section, we briefly analyze the traditional problems between SDEs and Man-Machine Interfaces. By analyzing this relationship, factors related to increasing software development productivity and the basic MMI design concept for SDE are determined.

Traditional SDEs problems

To increase software development productivity, several SDEs and methods have been derived [4,5,6], for example, (1) UNIX PWB (programmer's work bench), (2) remote job entry, remote-log-in, source code management, text processing, target machine development simulator, and (3) separation of software development, testing, and management environment[7]. These works have reduced software development efforts and increased development productivity to some degree. However, unsolved problems are piling up:

- (1) It is hard to use or refer to different information simultaneously,
- (2) It is troublesome to send or receive messages among various software programs,
- (3) It is difficult to extract and represent software development knowledge and know-how from experts, and to reuse this expertise,
- (4) It is hard to customize SDE MMI for each software development project,
- (5) There are many miscellaneous software development tasks, such as document management, trouble reports, and requirement reports,
- (6) Software can be very complex for its manipulation.

These problems were analyzed from the viewpoint of increasing software development productivity. The factors that influence productivity have been organized as follows:

Factor 1 Parallel execution of software development processes. E.g.: Under multi-task and multi-window environments, users communicate with several software tools simultaneously.

Factor 2 Automating regular software development processes, e.g.: Writing regular software development process for experts in a particular language and having novices reuse it.

Factor 3 Visualization of software development processes, reducing miscellaneous software development tasks, e.g.: Window, menu-driven, iconic interfaces for software development tasks.

MMIs and factors of increasing software development productivity

To provide automatic and parallel execution of software development process environments, and accumulation of software development processes and know-how, it is necessary to utilize WS MMI, such as windows, mouse, and pop-up-menus, systematically. Moreover, it is necessary to make full use of the

HandS Scenario. The basic design concept of the HandS-MMI system is indicated in Table 1. The H-Scenario and examples are presented in detail in sections 3 and 5, respectively. In large SDEs, project managers prescribe development process standards. In the HandS environment, project managers supply H-Scenarios as software development standards for each project member.

[Table 1] Factors: Increasing Software Productivity and Man-Machine Interface Primitives

Aim	MMIs
Parallel execution of software development process (Factor1)	Multi-Windows Multi-Tasks Pop-Up-Menus Icons Mouse
Graphic representation of software development process (Factor3)	
Reuse mechanism of text information (Factor2)	Rich and Powerful ScreenEditor Virtual Paper
Automating regular software development process (Factor2)	HandS System Scenarios Project Scenario definition User Scenario definition
Accumulation and reuse mechanism of software development process and know-how (Factor2)	
Personalizing each user software development environment (Factor 2)	
Decreasing miscellaneous jobs of software development (Factor3)	E-mail Office Automation Tools (e.x. DBMS)

HandS-MMI

In this section, special HandS features are described.

HandS Scenario (H-Scenario)

One of the new features of HandS is the H-Scenario. H-Scenario specifies the control flow of the software development process and its MMI by means of specific algorithmic descriptions. The encoding of the software development process is known as H-Scenario. Although the H-Scenario is slightly like traditional command procedures such as csh or Ksh in

the UNIX environment [8], it specifies the software development processes.

In software development, many kinds of knowledge has been needed. For instance, system designers had to have a knowledge of the target business. Programmers and operators had to have a knowledge of programming languages, usages of software development assistance tools, and so on. H-Scenario aims at embedding software development knowledges, which are necessary to develop software systems, in software development processes standards with useful menus and guidance man-machine interfaces. In large scale software development, there are many development processes standards including design method, development method, test method, documents formats, and software component name definitions. H-Scenario standardizes the software development processes, which have been depended on each project member, using the particular description language and functions. Then H-Scenario provides automatic execution of software development processes.

Using conventional command procedures, we cannot write scripts which include sophisticated MMI operations. However, in HandS environments, in addition to writing software development process control flows like traditional command procedures, we can take in various kinds of H-Scenario functions (Table 2). As it appears that menu-driven, iconic, mouse and multi-window interfaces are very useful for improving software development [9,10], MMI manipulation functions (such as windows, menus, and icons) are incorporated into H-Scenario functions. Also, since treat text information is often handled in software development, the screen editor control mechanism in H-Scenario is incorporated as well. Users write their own H-Scenario with H-Scenario description language. This language specification is a subset of C

HandS Editor

The HandS system is based on a full screen oriented editor. The HandS editor has two major features. First, it provides the reuse of text information. Every window is based upon virtual paper (see Figure 3). Thus, all editing commands are available in every window. Virtual paper is a mechanism which saves all input command histories and output data. Second, the HandS Editor provides defining or programming editor commands (**Package**). The H-Scenario mechanism can control the editor Package from each H-Scenario. Since HandS has a special PIPE which connects the editor Package to standard input/output of software development tools, a systematically unified Man-Machine Interface with end users can be provided by combining software development tools and editor Packages as front-end I/O. Package and its Scenario examples are presented in section 5.2.

[Table-2] HandS Scenario Functions

Category	Scenario Function name	Functions
Window Control	CreateWindow	Create logical window
	OpenWindow	Display window on the screen
	CloseWindow	Close window
	DeleteWindow	Delete window
Input/Output Control
	HomeClear	Erases the window & locates the cursor at home position
	DoHome	Locates the cursor at home position
	LoadFont	Font loading
Command Control
	System Batch	UNIX command exe.
	Execute	Submit batch job
	...	SubScenario exe.
Menu/Icon Control
	SetMenu	Menu registration
	UseMenu	Display the menu
	SetMenuItem	Menu item regist.
String Manipulation
	SetIcon	Icon registration
	DisplayIcon	Display the icon

Simple Graphics	StringIndex	Search string and return its position
	StringLength	Return string length

	DrawLine	Draw line
Interaction Control	DrawArc	Draw arc

	Pipe	UNIX tool execution with data pipe
	GetString	Read a line
Editor Control	WritePipe, ...	Write string to pipe, ..
	LoadPackage	Load the HandS editor package
	RunProcedure	Editor package execution, ...

communication control, others
	LoginHost	Log into host system
	LogoutHost	Log out host system

HandS Configuration

In this section, contrivances to materialize HandS futures, which are described in the section 3, are presented.

Network Configuration

HandS network configurations are shown in Figure 1. The HandS basic configuration consists of several bit-mapped terminals, PCs, and one UNIX environment. This HandS environment is used as a front-end system of mainframe computers. Bit-mapped terminals and PCs are connected to the UNIX environment by serial line interfaces such as RS-232C or RS-422A. In this study, we use bit-mapped terminal with 68000 CPU, 2 Mbyte local memory, and firmware (local window manager). We also developed a PC local window manager which emulates bit-mapped terminals. UNIX System V or 4.2/3 BSD running on a system similar to the Vax 11/780 was used. One of the important features of this system is it provides common MMIs for any HandS users with bit-mapped serial line terminals. Instead of using ordinary TTY terminals, we use bit-mapped terminals in which the firmware can accommodate window regions, mouse events, font-data, and local menus. Therefore, we also designed a vertically distributed environment between UNIX and bit-mapped terminals. The bit mapped side accommodates:

- (1) display region and local menu control,
- (2) local window management (move, reshape, top/bottom, create),
- (3) keyboard and mouse event control, and
- (4) font control.

The HandS on the UNIX System side accommodates:

- (1) full screen editor,
- (2) process and window management,
- (3) H-Scenario,
- (4) communication with the host computer, and
- (5) utilities.

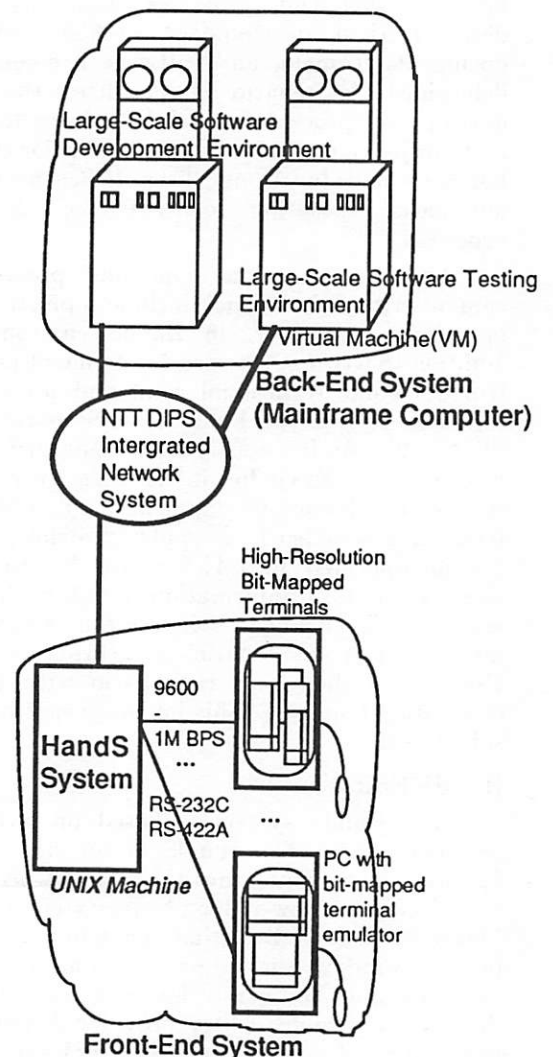
HandS Software Organization

HandS operates in UNIX environments (see Figure 2). The reasons for developing the empirical system in the UNIX environment are summarized as follows:

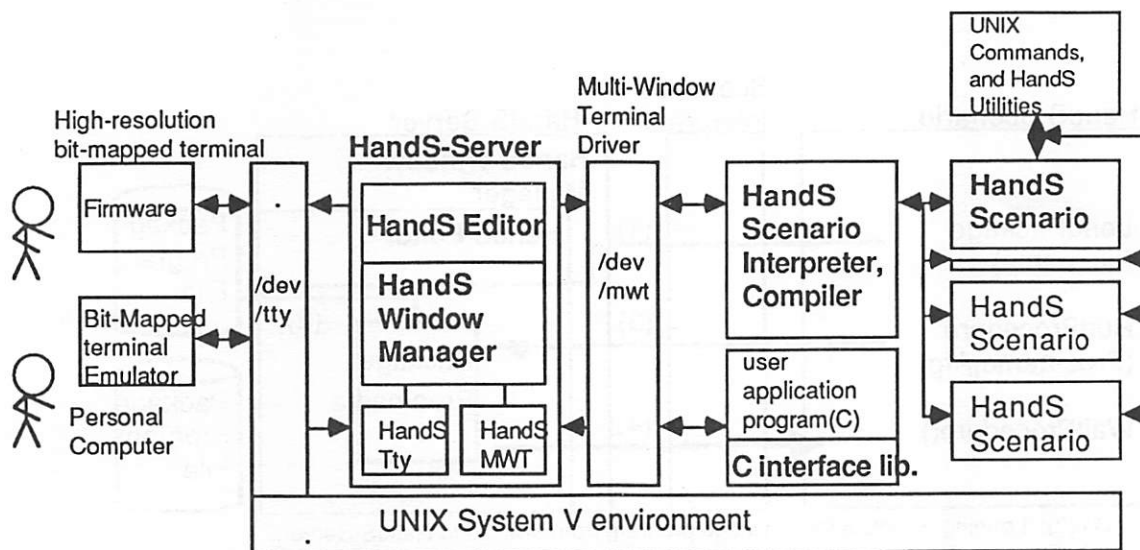
- (1) There are many useful development facilities, for example source text handling tools, for constructing new specialized systems.
- (2) The UNIX system is widely used, and it is relatively easy to transfer this specialized system to other hardware environments.
- (3) UNIX will become more widely used, especially in the personal computer world.

On the other hand, we did not use a bare UNIX system because UNIX environments are not suitable for novice users. Therefore, developing tractable MMI

for software development environment in the UNIX system may be the best methods. The HandS system consists of a **HandS-Server** (HandS window manager: **H-WM**, HandS-editor: **H-Editor**), **Scenario interpreter**, multi-window terminal driver, multiple H-Scenarios, HandS utilities, and firmware on bit-mapped terminals.

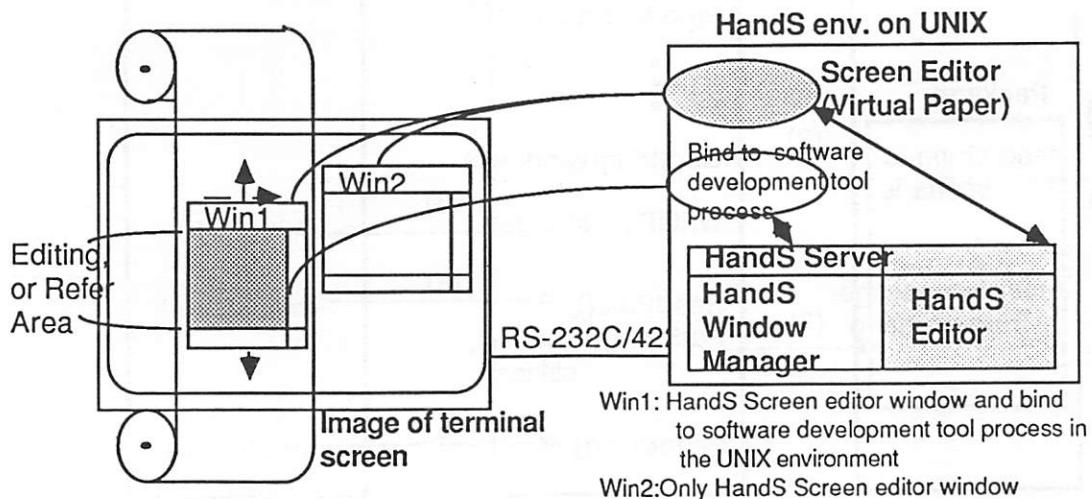


[Fig. 1] HandS for the software development environment

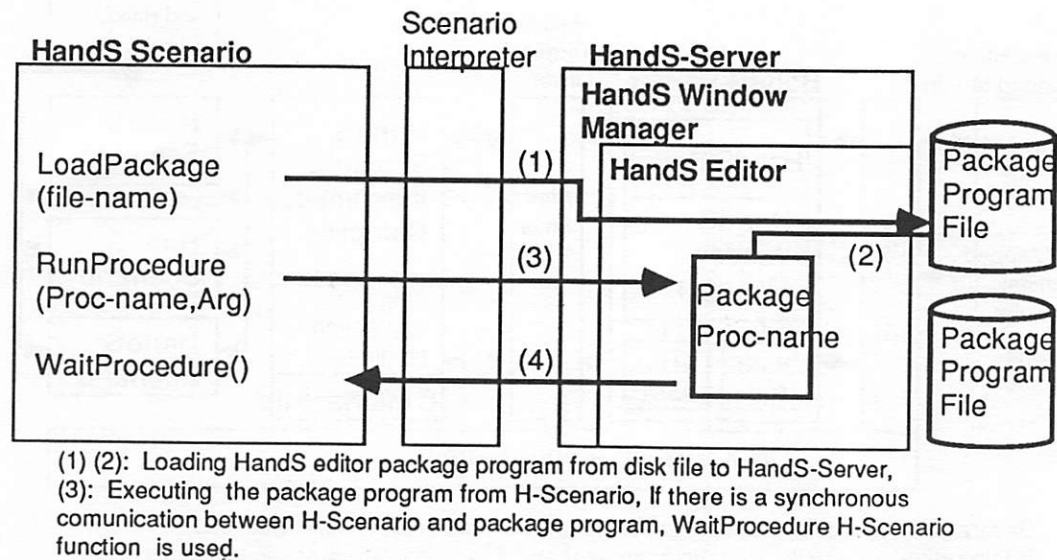


Several H-Scenarios are running under HandS-Server environment. H-Scenario is interpreted by HandS Scenario interpreter and Scenario interpreter sends instructions to HandS-Server. If there are any instructions to firmware, for example display window, then HandS-Server sends instructions.

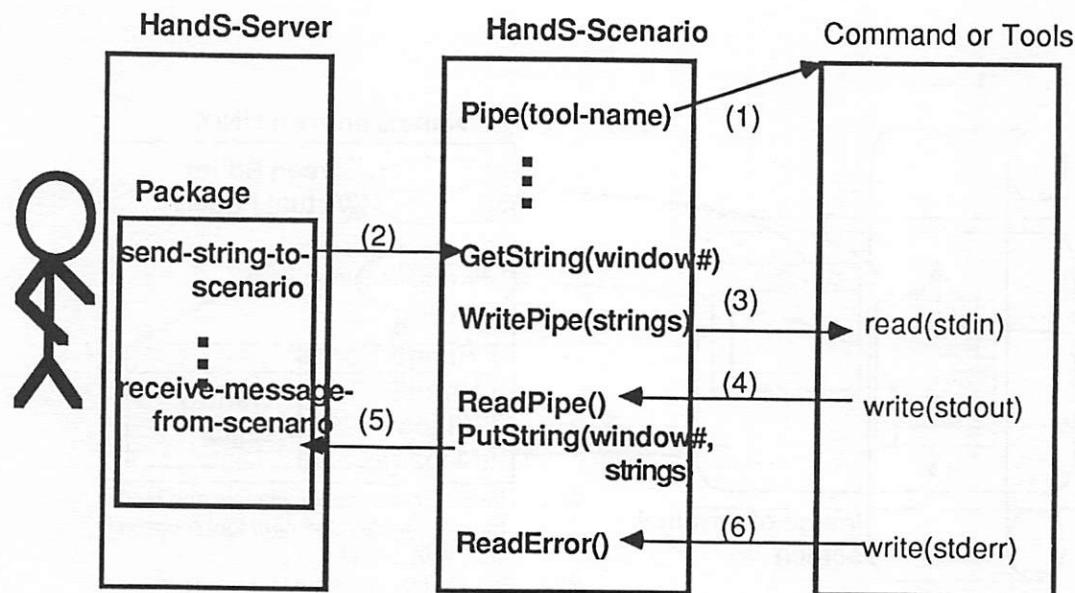
[Fig.2] HandS Software Configuration



[Fig.-3] Relationship between Window Manager and Editor Buffer



[Fig.-4] HandS Package program execution from Scenario and its control mechanism



(1) software development assistance tool and command execution instruction(Pipe), (2) package has strings, for example user input data, then send strings to H-Scenario, in H-Scenario receive these strings(GetStrings). (3) in H-Scenario, processing this string, then send to tool, (4) H-Scenario receives tool output data using ReadPipe scenario function, (5) Sends strings to Package, (6) H-Scenario also receives error message from tool by ReadError.

[Fig.5] I/O interface among Package and Scenario and tool

HandS-Server: Window Manager

The HandS window manager administrates the control data table for multi-windows (overlapped windows) and for processes which are activated from H-Scenarios. In the HandS system, every window is not always bound to these UNIX processes. There is a one-to-one correspondence between windows and H-Editor buffers. The window that is used for text processing only is not bound to these UNIX processes (see Figure 3). H-WM takes care of control data for each window, such as window #, window size, status (whether keyboard is assigned), state attribute (whether window moving, window reshaping, and keyboard assign are ok). For example, if windows are to be displayed in H-Scenario, H-WM creates logical windows and sends window displaying instructions to the bit-mapped terminal. Conversely, firmware sends instructions for executing particular tools to H-WM when end user select the tool execution menu item.

HandS-Server: H-Editor

(1) Basic functions

The HandS editor is a real time screen oriented editor, the basic editing functions of which are similar to EMACS [11]. For example, these include cursor movement, reconfigurable keymaps, creating new editing macros, and insert-mode editors. With the HandS editor, moreover, each editing buffer is bound to a window-region which is controlled under H-WM. With this binding of buffers and window areas, HandS makes using editor commands in all windows possible. For example, users can use editor commands, such as move cursor and search procedures in TTY windows (e.g. UNIX csh windows).

(2) Package

H-Editor and H-WM are developed in a modular fashion and are composed of many independent functions. The user extends H-Editor by writing their definitions in the some simple language used to write H-Scenarios. Programmed definitions are called **Package**. This simple definition language is a subset of C language. H-Editor offers the following functions as built-in commands of the Package program:

- (a) HandS editor buffer control,
- (b) display screen,
- (c) keymap definition,
- (d) output message control,
- (e) file manipulation,
- (f) string manipulation, and
- (g) text editing.

Package program consists of these built-in functions. In other words, users can write their own text-processing packages (for example, a syntax oriented template package) using primitive functions of the HandS-Server. Moreover, users can call and execute these packages as a front end of the ordinary software development tool in H-Scenarios.

(3) Package control

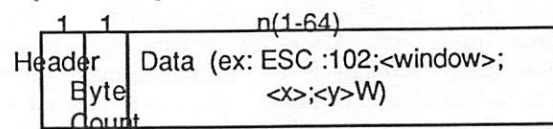
Package is not executed by itself. Packages are always activated by direct H-Scenario instructions. Packages are stored in disk files. Therefore, before executing Package, H-Scenario should load Package to the HandS environment. To activate Package and control it synchronistically, HandS has *RunProcedure* and *WaitProcedure* Scenario functions. The relationship between H-Scenario and Packages indicated in Figure 4.

(4) I/O interface between software development tool processes and Scenario and Package

If there is a necessity to develop a particular H-Scenario which has interactive communication between software development tool processes and Scenario, and if it is also necessary to use Package for I/O in the front end of software development processes, a method for binding software development process standard I/O (stdin, stdout, stderr) to Scenario is required. There are several methods to bind standard I/O to tool processes, for example, IPC (Inter Process Communication) and PIPE. PIPE is used in our empirical environment, because with it I/O interface between tool processes and Scenario can easily be developed. HandS has six Scenario functions as UNIX process and Package control functions. The control mechanism between tool processes, Scenario and Package is presented in Figure 5.

H-Scenario Interpreter

The H-Scenario-Interpreter interprets H-Scenarios and executes them. It then translates each H-Scenario function into H-protocol packets defined in the HandS environment. The H-protocol packet form is presented in Figure 6. In the HandS system, each module, such as the H-Scenario Interpreter, H-WM, H-Editor and firmware, communicates using this H-Protocol packet. In the HandS environment, ANSI ESC private sequences are used for data representa-



Header:Window#, or slave# of mwt
Byte Count:# of byte

[Fig.6] MultiWindow Protocol Packet

tion in the H-protocol. The HandS system also has an H-Scenario compiler to provide an efficient H-Scenario execution environment.

Other Modules

A special driver (/dev/mwt) for implementing multi-window environments on the UNIX system was developed. The HandS system also has other utilities such as a C interface library, IconEditor, ImageEditor, DirectoryEditor and host language syntax oriented checker.

Software development process assistance H-Scenario

In this section, usages of H-Scenario functions and H-Editor packages are described.

Basic H-Scenario

H-Scenario for tool execution

An example of the software development process is shown in Figure 7(a). This consists of displaying icons, allowing the end user selects icons, executing the corresponding tools such as *csh*, *H-Editor*, *direct-Scenario Interpreter* and *ImageEditor*.

Figure 7(b) and Figure 7(c) present the real H-Scenario list and its screen, respectively. The HandS system has two fixed windows, the *input window* and the *status window*. HandS writes the status messages of H-Scenario execution in the status window. The input window is used for particular editing command entry and strings for the HandS editor. H-Scenario in Figure 7(b) is divided into two major parts:

- (1) Icon-data registration (*SetIcon*) and screen display (*DisplayIcon*),
- (2) Waiting for user icon selection by mouse (*GetIcon*) and executing the sub H-Scenario corresponding to the selected icon, asynchronous mouse event processing routine.

Create and Display window

Figure 7(d) shows the basic H-Scenario for window creation and display, and Figure 7(e) shows the UNIX tool (for example, *csh*) execution scenario. In Figure 7(d), the H-Scenario function *CreateWindow* creates a logical window in the HandS-Server, and *OpenWindow* displays the real window on the bit-mapped screen. HandS-Server sends the display window instruction to firmware when HandS-Server receives the *OpenWindow* function from the H-Scenario interpreter. After the creation and display window, *SwitchWindow* is used for changing standard output to the target window and *Attach* is used for binding the keyboard to that window. If the top of the window area (Title bar: In HandS, users can write any information in this area, for example file/current directory name, window status) is black, the keyboard is assigned to that window. The */bin/csh* execution is shown in Figure 7 (e). First of all, H-Scenario interprets the creating window sub H-Scenario (*include* function), and then executes the UNIX tool (*system* ("*/bin/csh*") ;). Eventually, it will be possible to write software development processes with MMI manipulation such as H-Scenario easily and execute them at once without compiling and linking them with the MMI library.

Menu description

A simple menu-driven Man-Machine Interface example is presented in Figure 7 (f). This is a menu interface H-Scenario, which is a rewritten version of the iconic interface H-Scenario (Figure 7(b)). *SetMenu* defines the number of menu items.

SetMenuItem

refers to the menu item registration and *UseMenu* displays the menu on the screen and waits for user selection.

In this section, tool execution, window control, and menu control H-Scenarios are presented. These basic H-Scenarios are provided as components. Users can modify them and use them in their own H-Scenarios. The HandS system has several H-Scenario functions (Table 2). Besides MMI manipulation H-Scenario functions, HandS has very useful H-Scenario functions for large-SDEs. For example, string manipulation functions for interpreting tool messages on main-frame computers, and synchronous tool execution control for parallel execution of software development processes.

HandS Scenario with H-Editor Package

An H-Scenario with H-Editor Package as an input/output front-end for software development tools is shown in Figure 8. Assume "Sample-package" is loaded into the HandS-Server environment by the *LoadPackage* function and gets argument "file-name" for the software development tool in the H-Scenario List. Also suppose that this Package displays the data entry template and receives user input. In the H-Scenario List, MMI descriptions such as window operation and menu-driven interface are omitted.

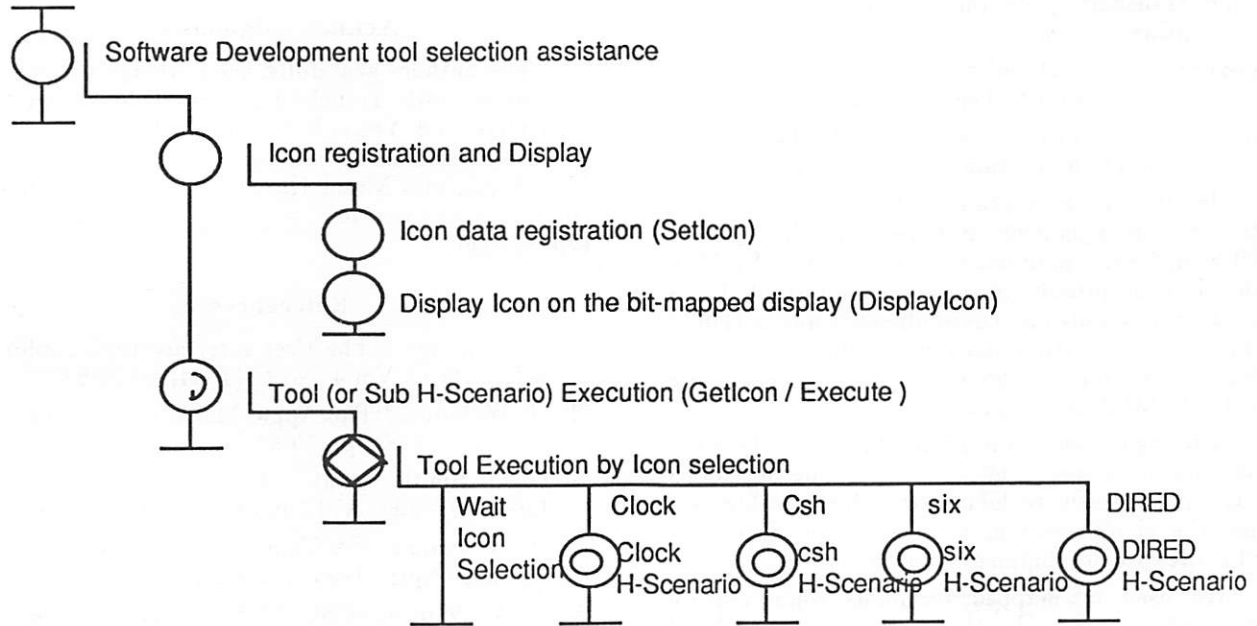
The H-Scenario function *Pipe* executes the software development tool. Then HandS-Server creates a pipe between H-Scenario and the tool. Next, *RunProcedure* executes the H-Editor Package in the window #. When the Package is activated by H-Scenario, in this example, users can enter the data with the Package data entry template (e.g.: data entry guidance). After Package gets data from the user, it sends the data string to H-Scenario by using H-Editor built-in commands (*send-string-to-scenario*). At this time, H-Scenario function *GetString* is waiting for the data from Package. Finally, H-Scenario receives the data from Package (and generates new string-data based on the received data), and sends the new string-data to the software development tool (*WritePipe*).

In this manner, H-Editor Package turns out the user input/output front-end for software development tools to improve MMIs. HandS-Server and H-Scenario provide these useful facilities for software development task assistance.

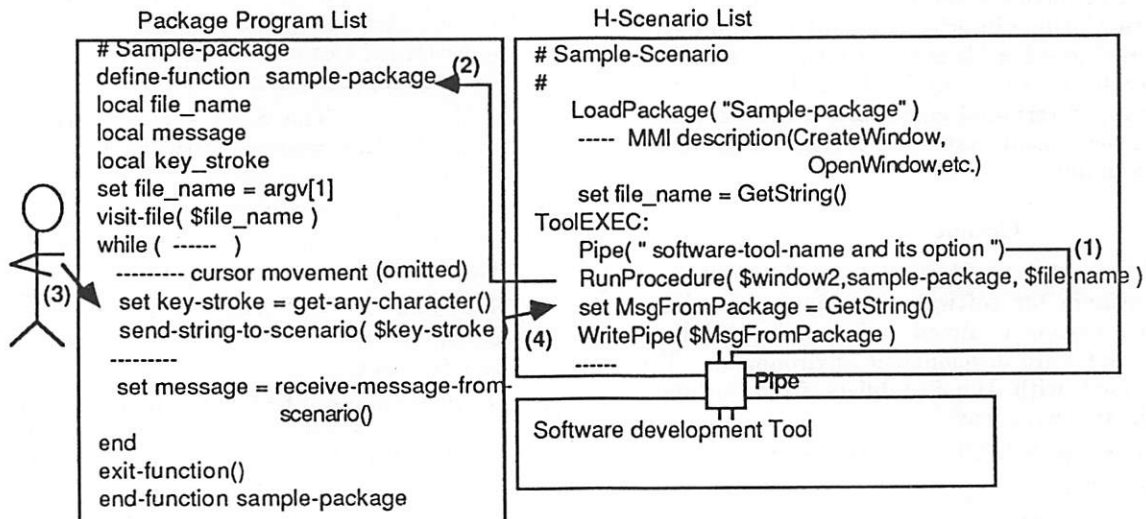
HandS Utilization and Future

In this section, the HandS utilization as a large scale SDE on a host computer is presented. To decrease software development tasks and increase the efficiency of software development in the mainframe environment, HandS will provide the following facilities besides H-Editor handle to program source text and check syntax:

- (1) syntax oriented editor, for example Ada and other



[Fig.7(a)] Software Development Assistance H-Scenario
(Tool selection assistance H-Scenario : Iconic Interface)
(HCP Chart description)



(1) Execution of Interactive software development tool, at this time, HandS-Server makes a Pipe between H-Scenario and tool. (2) Execution of HandS editor Package as a front end of software development tool, (3) User inputs data under Package program interface, (4) Then Package sends user input data to H-Scenario. (5) H-Scenario send this user input data to software development tool by Pipe.

[Fig. 8] Example of Package program and Scenario

programming languages, to decrease program source text editing tasks,

- (2) file transport program between heterogeneous operating systems,
- (3) execution control and visualization of tool execution in a host or a VM environment.

These facilities will be located in the UNIX environment and used as a programmer's work bench.

HandS uses a communication system between heterogeneous operating systems (UNIX and the NTT mainframe computer operating system BEAM), which is conceptually congruent to ISO/OSI. Users log into the mainframe environment using a HandS window as a virtual terminal, and control the software development tool on the host system under the HandS-MMI environment.

Although communication facilities between UNIX and host operating systems is being developed at the NTT software laboratories for possible systems, the HandS system is presently used for the UNIX software development environment.

We used bit-mapped terminals connected by serial line interface. In these configurations, MMI operation response time depends on firmware performance and transmission speed between firmware and the UNIX system. Since we designed the firmware in a modular fashion to decrease data transmission with the HandS-Server, we can achieve high performance even in prototype versions.

In our first empirical environment, we used bit-mapped terminals. However, personal computers are widely used in software development. To provide the HandS Man-Machine Interface to many PC and UNIX users, we developed a bit-mapped terminal emulator which operates on PC-DOS. With the HandS system and bit-mapped terminal emulators, a less expensive software development system with rich and powerful MMIs is available.

Conclusions

In this study, we described the Man-Machine Interface system for software development environments. This system is aimed at improving MMIs of large scale software development environments. The SDE equipped with HandS differs from ordinary SDEs in the following ways:

- (1) H-Scenario with MMI description and screen editor Package,
- (2) Server-programmable (Package) environments, Package can be used as input/output front end system for software development tools,
- (3) HandS is used for improving MMIs of large SDEs
- (4) HandS provides a window system by serial line interface.

These HandS facilities are useful for parallel execution, visualization and automation of software development processes. However, new methods of

improving Man-Machine Interfaces for large software development environments will become necessary.

Acknowledgments

The authors gratefully acknowledge the helpful discussions with Youichi Itoh, Syunichi Fukuyama, Yuji Ono, and Yosinob Masatani. Acknowledgment also is due to Masahiro Shimomura, Kaoru Mitsuhashi, and Motoi Karakawa for their valuable comments and for their help in producing the prototype system.

References

- [1] B. A. Myers, "The User Interface for Sapphire," IEEE CG&A Vol.4, No.12 (1984) pp.12-23.
- [2] G. Williams, "The Apple Macintosh Computer," Byte, Feb. 1984, pp.30-54.
- [3] D. C. Smith, et al., "Designing the Star User Interface," Byte Vol.7 No.4 (1982) pp.242-282.
- [4] W. Teitelman, "A Tour Through Cedar," IEEE Software April (1984) pp.44-73.
- [5] T. A. Dolotta, et al., "UNIX Time-Sharing System: The Programmer's Workbench," The Bell System Tech. Journal, Vol. 57 No.6 (1978).
- [6] H. Nagano, et al., "Scenario", an Inducement and Enforcement Mechanism for Software Engineering Standards (in Japanese)," Electrical Communications Labs. Tech. Jour., NTT Vol. 34, No.6 (1985) pp. 927-939.
- [7] T. Satoh, et al., "An Integrated Software Development System for Data Communication Systems Utilizing Various Types of Computers," Review of the Electrical Communication Labs., NTT Vol.33 No.3 (1985) pp.464-472.
- [8] D. G. Korn, "The Shell - Past, Present, and Future," UNIX System Software Tech. Seminar, Tokyo (1986).
- [9] J. D. Foley, et al., "The Human Factors of Computer Graphics Interaction Techniques," IEEE CG&A Vol.4 No.11 (1984) pp.12-48.
- [10] S. Fleischman, "An Icon Oriented Interface for a UNIX Workstation," IEEE Workstation Tech. and System Conf. (1986) pp.76-83.
- [11] R. M. Stallman, "EMACS: The Extensible, Customizable, Selfdocumenting Display Editor," ACM SIGPLAN SIGOA symposium on Text and Manipulation, June (1981) pp.147-156.
- [12] S. Hanata, et al., "Documentation Technology for Packing Hierarchical Function, Data and Control Structures," Proc. of COMPCON Fall (1981) pp.284-290


```

#           HandS Scenario1: Tool selection assistance
#           H-Scenario
Echos( "\nHandS software development tool selection scenario" )
set int_stack = ( ) set job_stack = ( ) set icon_stack = ( )
Echos( " ... Now loading tool icon! Wait a minute! " )
set I_file = ( clock Csh Scenario Dired )
foreach arg ( $I_file )
    set file = "/usr1/HandS/Icons/System/$arg"
    set icon_no = SetIcon( "$file" )
    set Icon_stack = ( $icon_stack $icon_no )
end # Display Icon
DisplayIcon( 0, $icon_stack[1], $I_file[1], 20, 620 )
DisplayIcon( 0, $icon_stack[2], $I_file[2], 20, 550 )
DisplayIcon( 0, $icon_stack[1], $I_file[1], 20, 480 )
DisplayIcon( 0, $icon_stack[1], $I_file[1], 20, 410 )
# Wait for tool selection and Tool execution
while (1)
    Echos( "\nPlease select Tool Icon!" )
    if ( $#int_stack != 0 ) then
        set cid = $int_stack[1] shift int_stack
    else
        set cid = GetIcon()
    endif
    switch ( $cid )
        case $icon_stack[1]:
            set job_id = Execute(clock) breaksw case
        $icon_stack[2]:
            set job_id = Execute(Csh) breaksw case
        $icon_stack[3]:
            set job_id = Execute(six) breaksw case
        $icon_stack[4]:
            set job_id = Execute(Dired) breaksw
    endsw
    if ( $?job_id == 1 ) then
        if ( $job_id != "-1" ) then
            set job_stack = ( $job_stack $job_id )
        endif
    endif
    ReverseIcon( 0, $cid )
end
# For asynchronous Icon click processing
clock:    set ID = $icon_stack[1]
          goto Click Intr
Csh:      set ID = $icon_stack[1]
          goto Click Intr
Scenario: set ID = $icon_stack[1]
          goto Click Intr
Dired:    set ID = $icon_stack[1]
          goto Click Intr
Click Intr: set int_stack = ( $int_stack $id )
ExitInterrupt()

```

Figure 7(b): Software development assistance H-Scenario List

```

# HandS Scenario2: Window Creation and Display
#           H-Scenario
#   NTT Software Laboratory, Tokyo   Japan
#   /usr1/HandS/Scenario/System/create.sc
#
#< input interface > w_argv : window attributes
#< output interface > w_argv[1] : window#

```

```

#
set w_argv[1] = CreateWindow( "$w_argv[9]",
    $w_argv[4], $w_argv[5], $w_argv[6],
    $w_argv[10], $w_argv[8] )
if ( $w_argv[1] == "-1" ) then
    Echos( "\nCan't create window!" )
    goto Error_return
endif
#
set status = OpenWindow( $w_argv[1],
    $w_argv[2], $w_argv[3] )
if ( $status == "-1" ) then
    Echos( "\nCan't open window $w_argv[1]" )
    DeleteWindow( $w_argv[1] )
    goto Error_return
endif
#
set status = SwitchWindow( $w_argv[1] )
#
set status = Attach( $w_argv[1] )
#
goto Return
Error_return:
set w_argv[1] = "-1"
Return:

```

Figure 7(d): Window Creation and Display H-Scenario Example

```

# HandS Scenario3: UNIX Command and User Tool
#           Execution H-Scenario Example
#           NTT Software Laboratory, Tokyo Japan
# /usr1/HandS/Scenario/System/csh.sc
#
set iconfile = "/usr1/HandS/Icons/System/cl_Csh"
set Icon_no = SetIcon( $iconfile )
#
location = ( 300 100 ) , width = 724, height = 438,
# GO = 16, delete-attribute = no, title = "/bin/csh"
set w_argv = ( -1 300 100 724 438 16 0 n "/bin/csh"
    $icon )
include /usr1/HandS/Scenario/System/create
#
if ( $w_argv[1] == "-1" ) then
    Exit()
endif
setenv WINDOW $w_argv[1]
#
System( " /bin/csh" )
#

```

Figure 7(e): UNIX tool execution H-Scenario example

```

# HandS Scenario4: System Tool Menu H-Scenario
#           NTT software laboratory, Tokyo Japan
# /usr1/HandS/Scenario/System/ToolMenu.sc
Echos( "\nHandSStartup Scenario" )
set job_stack = ( )
set Icon_no = SetIcon(
    /usr1/HandS/Icons/System/HandS )
#
set MenuHeader = "HandS System Tool Menu"

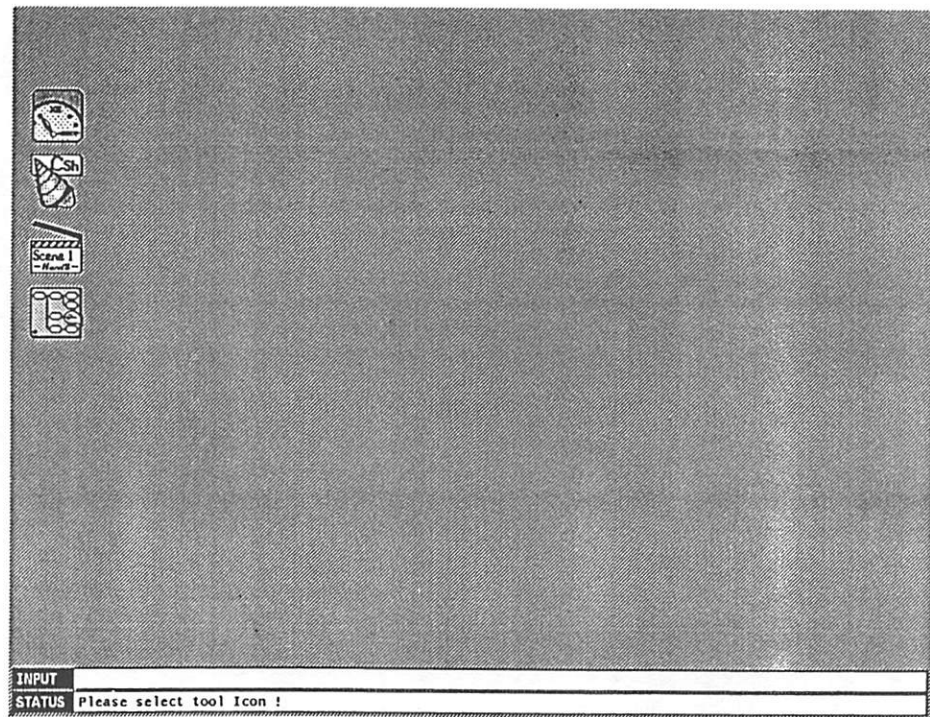
```

```

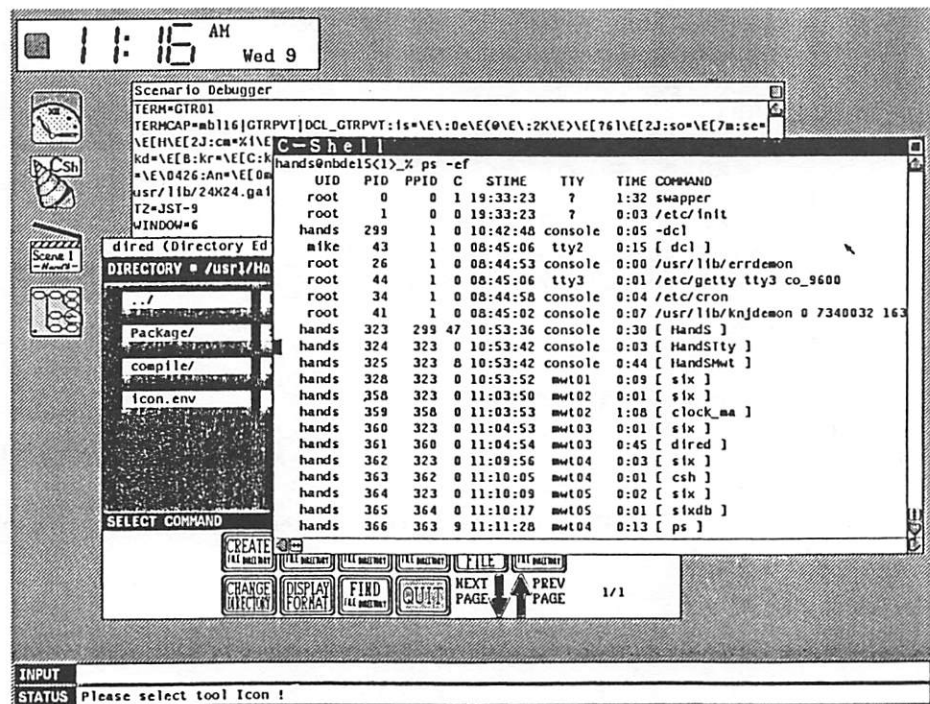
    set MenuHeaderSize = StringLength(
        "$MenuHeader" )
    set MenuItem = ( \
        "      Clock      "\
        "      C-shell    "\
        "  Scenario Debugger "\
        "  Directory Editor "\
        "      Quit      "\ )
# Menu registration
    set Menu_no = SetMenu ( $#MenuItem, 16,
        $MenuHeaderSize )
    SetMenuItem ( $Menu_no, 0,"$MenuHeader",y,y )
    set position = 1
    foreach arg ( $MenuItem )
        SetMenuItem( $menu_no, $position,
            &MenuItem[$position]",n,y )
        @ position++
    end
# Display Icon and Menu and Tool execution
    DisplayIcon( 0, $icon_no , "", 20,620)
    while ( 1 )
        Echos( " \nPlease select Icon!" )
        set Menu = UseMenu( $Menu_no, 20,620)
        EraseMenu ( $Menu_no )
        switch ( $Menu )
            case1: set job_id = Execute(clock)
                    breaksw
            case 2: set job_id = Execute(csh)
                    breaksw
            case 3 :set job_id = Execute(six)
                    breaksw
            case 4 :set job_id = Execute(DIRED)
                    breaksw
            case 5 :goto End
            default:
                Echos( "\nPlease select Menu Item again" )
                goto Cont
        endsw
        if ( $?job_id == 1 ) then
            if ( $job_id != "-1" ) then
                set job_stack = ( $job_stack, job_id )
            endif
        endif
        Cont:
        ReverseIcon(0, $icon_no )
    end
End:
DeleteIcon( $icon_no )

```

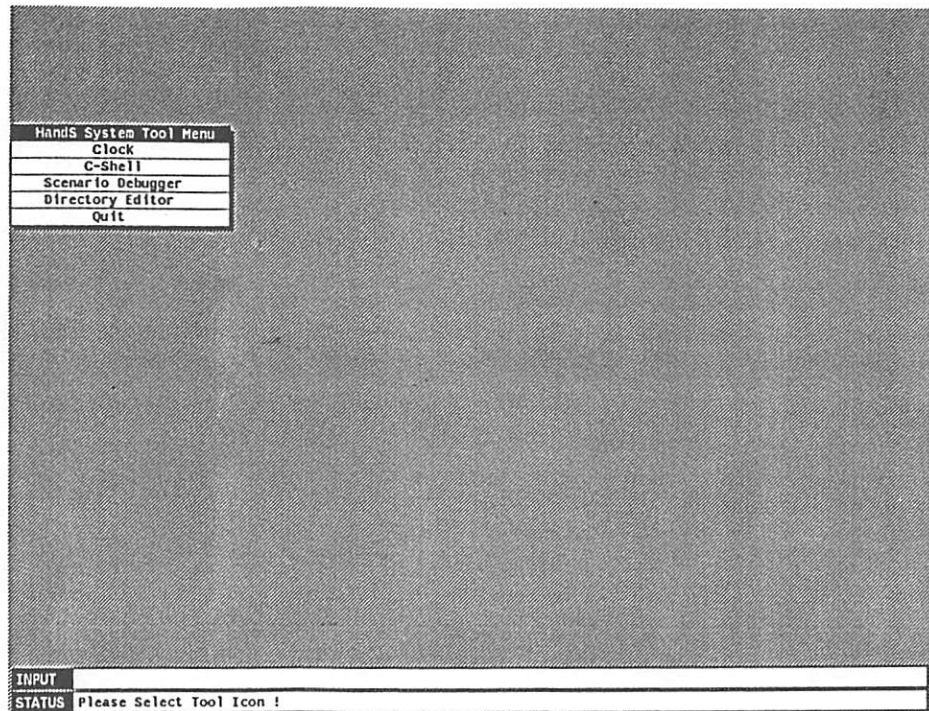
Figure 7(f) Menu-Driven interface H-Scenario example



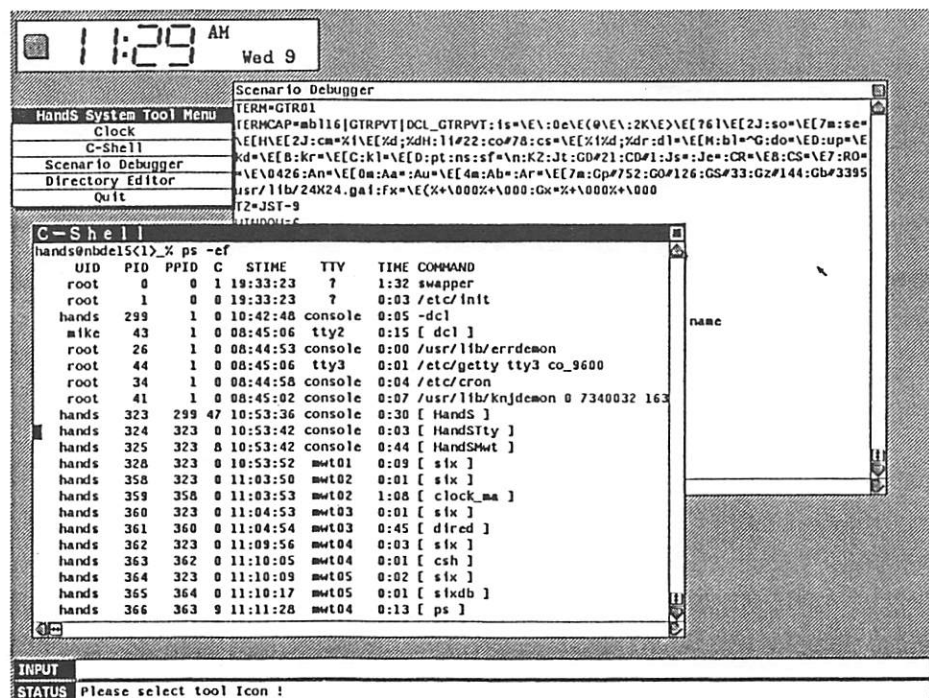
[Fig.7(c)-1] Software development assistance H-Scenario execution
(Waiting for icon selection)



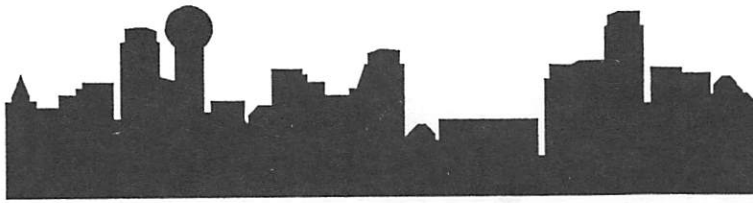
[Fig.7(c)-2] H-Scenario execution
(Execution of software development tools)



[Fig.7(f)-1] Software development assistance H-Scenario execution
(Pop-up menu driven MMI H-Scenario)



[Fig.7(f)-2] H-Scenario execution
(Execution of menu items)



USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

A Memory Allocator with Garbage Collection for C

Michael Caplinger
Bell Communications Research
Research, Room 2A-261
435 South Street
Morristown, NJ 07960
(201) 829-4635
mike@bellcore.com or {ihnp4,decvax}!bellcore!mike

ABSTRACT

The memory allocation primitives available in C (*malloc* and its associated routines) rely on explicit programmer effort to release unneeded storage. For many applications, it is difficult if not impossible to locate all the places storage must be released. A possible solution is automatic reclamation of storage, or *garbage collection*; languages like Lisp have provided garbage collection for over twenty years, but such facilities have been slow in coming in more traditional languages. Here, we describe the design, development, and performance of a garbage-collecting allocator for C. Based on a "conservative" approach, it requires no programmer intervention and can be inserted into existing programs without modifying their code.

Introduction

Dynamic memory allocation is a crucial facility for most realistic programs, which cannot know at compile time how much memory they will need, or in what forms. Object-oriented programming is even more demanding of dynamic allocation, as that paradigm encourages the frequent creation of new objects in memory. In addition, complex sharing of objects via pointer references makes it very difficult for programmers to know when an object can be explicitly freed. Failure to free allocated storage results in *storage leaks* that ultimately cause programs to exhaust their memory and crash when they have no real reason to. The easiest way to solve such problems is to provide automatic reclamation of unused storage (usually called *garbage collection*.)

Languages now being used for object-oriented programming (such as Smalltalk or Lisp) have memory allocation with garbage collection built in at a very low level. However, we wrote the Telesophy system [Caplinger], a distributed object-oriented information system, in the C language under the Unix operating system. Not only does C not provide garbage collection, but it rapidly became apparent that the standard memory allocator used in Unix C was inadequate to support our object-oriented programming style. In response, we set out to build a new allocator that addressed the problems.

(A note on terminology: we refer to each piece of memory allocated by a user request as a *segment*, and the portion of a process's memory space devoted to dynamic allocation as the *heap*.)

Design goals

Compatibility with existing *malloc*

The first and most important goal for the new allocator was that it be completely compatible with the existing Unix routines (*malloc*, *free*, *realloc*). This was important because we had already written most of the system using these routines, and had little desire to radically change existing code. In addition, many unrelated Unix library routines (such as the *stdio* system) used *malloc* and company, and we did not want the added complexity of making the new allocator coexist with the old.

Error resistance

We wanted to make the detection of memory reference errors easy. The Telesophy user interface program consists of a collection of applications structured as lightweight processes in the same address space, operating on a large number of data objects. Any illegal memory access (such as writing off the end of an allocated segment) might cause an error in a distant and unrelated piece of code, or it might cause the memory allocator itself to fail. The standard Berkeley 4.2 *malloc* is especially bad in this regard, for reasons we describe later. Since such problems had already caused considerable grief, we wanted to make them easy to find.

Space efficiency

The early virtual-memory versions of Unix encouraged programs to use a lot of memory, and the standard allocator strove to enhance virtual-

memory performance at the cost of being wasteful with space. Many of those motives no longer make sense. A typical Sun workstation used for Telesophy has eight megabytes of physical memory, and (because of frugal swap space allocation, especially for diskless machines) not much more virtual space. We can safely assume that paging is not a problem, so virtual-memory efficiency is not as important as simple space efficiency. Space efficiency remains important because we often have twenty or thirty server programs running on the same machine, and use of the earlier allocator caused all of them to use up huge amounts of virtual memory (for no very good reason) and run their host machine out of resources. We describe the standard allocator's problems in this regard in more detail in the next section.

Garbage collection

As mentioned in the introduction, our object-oriented programming style made it difficult or impossible to determine if a particular object's memory could be freed. In addition, even when it was possible it was simply too painful to track down all the possible deallocation points. This caused a large number of problems in programs that had to manipulate a huge amount of data (our database import programs regularly process hundreds of megabytes in a single run) or that had to run indefinitely (some of our server programs ran out of memory after a few hours of heavy usage and had to be restarted.)

Effective explicit free

In contrast to the last goal, we wanted the explicit freeing to be effective when it *was* used. Many memory allocators with garbage collection provide no way to explicitly free an object, and end up spending too much time collecting garbage. We hoped that by freeing as much storage as we could conveniently, we could cut down on the frequency of garbage collection.

Deficiencies of the standard allocator

The Unix Version 7 allocator used a first-fit linear search of all segments, and thus in the worst case had to look through all of allocated memory before expanding the address space. In the small address space of the PDP-11, this was never a problem, but when the VAX made large virtual address spaces common, the excessive number of page faults caused by the exhaustive linear scan became unacceptable. The version of *malloc* and its associated routines distributed in 4.2 Berkeley Unix and some of its derivatives has an allocation policy based on multiple powers-of-two free lists. The size of a segment request is rounded to the nearest power of two (minus header overhead) and the free list containing blocks of that size is searched. The intent of this strategy is to cut down on fragmentation (since blocks of roughly the same size come from the same space) and limit the

amount of memory that must be traversed.

However, this strategy suffers from several serious problems.

- It is quite wasteful of space, since segment sizes are always rounded up to a power of two.
- Because of the rounding, segments usually have large amounts of "dead space" at the end. A common program error is to run off the end of an allocated segment, but because of the extra space, this error often goes unnoticed.
- Because of multiple free lists, the allocator sometimes runs out of memory before it ought to: for example, if a block of size N is allocated and brings the process's virtual memory to its maximum, and then that block is freed, a request for a segment of $N/2$ will fail because the allocator will not look on the N free list for $N/2$ space.
- When *malloc* fails, it returns the same result as it does when memory is exhausted. There is no way to tell (without compiling with additional debugging, which requires source access) if memory was really exhausted or if one of the free lists was corrupted.
- In cases where memory has been badly corrupted by memory overwriting, the allocator crashes catastrophically.

In our new allocator, we wanted to avoid these problems.¹

Adding garbage collection

The essence of garbage collection is determining when a piece of storage can no longer be referenced by the program, and can thus be reused. A segment cannot be used if no pointers point to it, so garbage collection must somehow keep track of the correspondence between all the pointers in a program and all the allocated segments. There have been two approaches to this problem: mark/sweep algorithms, and reference counting [Knuth].

In reference counting, each segment contains a count of the number of pointers that currently point to it (the *reference count*.) The count is incremented when a new pointer is assigned to point to the object, and decremented when a pointer is cleared. When the count goes to zero, the segment can be reused (and any segments that segment references have their counts decremented, and so on.)

Obviously, maintaining the reference count requires us to perform operations at each pointer assignment and each time a pointer is cleared (as occurs when a routine defining a local pointer is exited.) The best way of doing this is to explicitly modify the language compiler to generate the needed reference count management. Requiring the programmer to add code to do this at every appropriate point

¹The version of Sun Unix we were using when we started the project used the BSD 4.2 allocator. Since then, Sun has rewritten *malloc* completely to add, among other things, extensive debugging capability.

is so error-prone that such a solution is probably worse than the problem was. Unfortunately, compiler modification was not within the scope of our project.

In the alternative scheme, mark/sweep, a single *mark bit* is associated with each segment. To do a garbage collect, the mark bits of all segments are first cleared. The collector then examines all the active pointers in the program and sets the mark bits of each segment they reference, as well as all the segments referenced by pointers within a marked segment, recursively. After this mark phase is complete, the system knows that any segment with its mark bit still off can be freed.

Again, mark/sweep requires us to know something — the location of all the active pointers in the program and in segments. Only the compiler has this information, and would have to be modified to communicate it to the garbage collector.

However, we can make a simplifying assumption and make the modifications unnecessary. Instead of checking only the pointers in a program, we check everything that *could be* a pointer by looking through all of memory. If a pointer candidate meets all the criteria of “pointerness,” the segment it references is marked. This approach has usually been termed *conservative garbage collection* [Boehm].

The extension of a compiler to add garbage collection to the Mesa language is discussed in [Rovner]. A reference-counting system that only uses knowledge of the location of pointers within heap-allocated segments is described in [Christopher].

The new allocator

We wanted to build an allocator which would solve all of the problems associated with the standard version, perform well in our environment, and make garbage collection possible. Adding garbage collection was more or less independent of any allocation strategy, but we wanted the result to be easy to develop and debug, so we fell back to the earlier Ver-

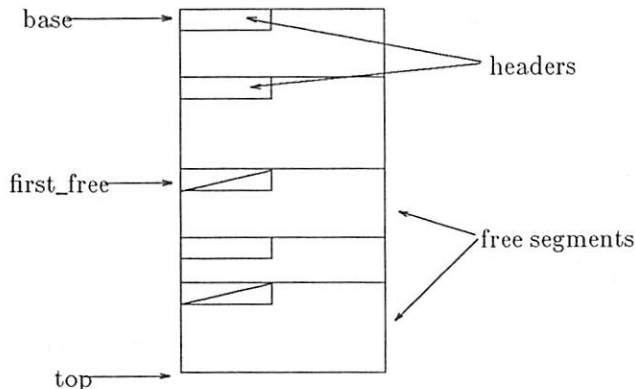


Figure 1: Memory Allocator Heap Layout

sion 7 policy and used a simple allocator with linear first-fit. (As mentioned before, we were unconcerned

about virtual memory performance.)

Figure 1 shows the layout of the space managed by the allocator. *base* points to the first segment in this space, *first_free* to the first free segment, and *top* to the first byte not yet in the program address space. Attached to each segment is an eight-byte header, defined by the following C structure:

```
struct header {
    unsigned magic : 32;
    unsigned inuse : 1;
    unsigned mark : 1;
    unsigned alien : 1;
    unsigned size : 29;
};
```

magic is a “magic number,” an unlikely bit pattern the allocator uses to verify if a given piece of memory is actually a segment header. The allocator maintains the magic number in each segment, whether it is free or in use. During all allocation activity, the system constantly checks the magic numbers of every segment it examines. If the number is found to be bad, the system aborts immediately by default. There is also a debugging function which verifies the integrity of memory by checking the magic number of every segment. Aborting on error detection seems like a harsh measure, but we can afford to have little tolerance for overwriting errors.

inuse indicates if the segment is free or allocated. The *mark* bit is used by the garbage collector, and the *alien* bit’s use is described later. Finally, the size of the segment is stored in *size*. The maximum segment size (and as a result the maximum heap size) is therefore 2^{29} , or about 536 megabytes. If this limit should become a problem, bits can be deducted from the magic number.

Allocation is simple. The requested segment size is rounded to the next multiple of four bytes (to insure that segments always lie on word boundaries) and the segment pointed to by *first_free* is checked to see if it is large enough. If it is, the segment is split and the first portion is allocated, with the leftover becoming the new *first_free* segment. (If the leftover is not large enough to become a new segment, it is appended to the new allocated segment instead, and *first_free* has to be located by linear search.)

If the first segment is not large enough, the system simply searches forward through all segments until it finds one large enough. If the entire space is searched without finding enough free storage, the memory space is expanded with the *sbrk* system call, and space is allocated from the resulting free segment. If the memory space of the process is exhausted, the system tries a garbage collection, as described in the next section.

Frees are implemented simply by setting the *inuse* bit of the freed segment to false, and updating *first_free* if necessary. As a partial optimization, if the segment immediately following the one being freed is also free, the two are merged. In general,

complete coalescence cannot be done because since the segment chain is only linked in the forward direction, the immediately previous segment cannot be located. Instead there is a separate routine to coalesce memory that is called by the garbage collector, and can be invoked explicitly by the programmer.

Note that this strategy is a slight variant of the Version 7 allocator. The pros and cons of this and a number of other allocation strategies are discussed in [Korn].

The garbage collector

The garbage collector is invoked either explicitly by the programmer, when the *sbrk* system call fails due to memory exhaustion, or when the allocator tries to use more than a preset amount of space. (The last condition makes it possible to limit memory usage to less than the system limits. The preset limit is considered soft, so the allocator continues to function after it is exceeded.)

The garbage collector uses the mark and sweep algorithm described before. First, it clears the mark bit in each segment header. It then examines the stack and data segments of the program, as well as the machine's registers², looking at every four-byte quantity on every two-byte boundary.³ The resulting candidates must meet the following conditions:

- (1) clear in the low two bits (since all segments are allocated on four-byte boundaries)
- (2) between *base* and *top*
- (3) pointing at a piece of memory that has a valid magic number.

These three tests are not completely sufficient to determine if a pointer references a segment, for two reasons. First, a piece of memory can masquerade as a segment if it contains a magic number. This is unlikely since the magic number bit pattern rarely occurs in other data, and the allocator scrupulously destroys magic numbers as segments are split and merged. A second case occurs when the pointer references not the first byte of a segment, but some later byte; we call such cases *interior pointers*. A segment which is referenced only by interior pointers will be erroneously freed.

There are two possible solutions to these problems, both of which add to the time or the space complexity of allocation. One is to traverse the entire segment chain to see if a pointer references the start or interior of an active segment; this would obviously be very time-consuming. A second solution is to maintain a separate table describing allocated segments

²The need to examine the registers is the primary barrier to portability; it requires a few dozen lines of assembly code. If it could be insured that all registers used by the program proper (as opposed to the garbage collector) had been saved on the stack, examining the registers wouldn't be necessary.

³This works on VAX and Sun hardware, but would have to be adjusted on machines with different pointer sizes or alignment requirements.

and back-link each segment to its entry in the table. A similar approach is described in [Boehm].

In our system, we live dangerously and assume that the three-part test is sufficient. Therefore, we cannot handle segments referenced only by interior pointers correctly; so far, these have not occurred in our code.

When we mark a segment, we recursively sweep through its contents looking for candidate pointers, and so on. Once a segment is marked, it is not swept further. This allows circular data structures to be garbage-collected. Note that this requires sufficient stack space to manage the recursion. Schemes that do not consume stack space are described in [Knuth], but are slower, more complex, or both; stack space is not much of a problem in our environment.

After all marking is complete, any segment not marked can be freed. This is done simply by setting *inuse* to false.

After all the unmarked blocks have been freed, memory is coalesced in a final pass, and *first_free* is updated. Coalescence merges all runs of contiguous free segments into a single large one. Note that this still might leave the heap quite fragmented. If we knew exactly where all the pointers in the program were, we could also move all the segments in use to the beginning of the heap, readjust all the pointers, and leave a single free segment with all available free space at the end of the heap. (This is commonly done in the *copying* garbage collectors found in some Lisp systems.) We are unable to do this in our allocator since we have no certain knowledge of where all the pointers are. While we only waste space by assuming that a value that resembles a pointer is a pointer, if we then modify that value we will be in serious trouble.

Experience

Alien segments

When we incorporated the new allocator in the Telesophy user interface program, a complex piece of window-based software consisting of about 15,000 lines of C code, we encountered an unexpected problem with the Sun window system. Right after the first garbage collection, we noticed that certain graphics operations performed by the Sun libraries, like text drawing and *bitblt*, began to fail in curious ways. Sometimes the entire workstation would crash with a kernel fault. We traced this problem to certain memory segments allocated early in the initialization of the Sun library. Apparently these segments were being garbage-collected while still in use.

We never determined exactly what was going on, since we have no access to the source in question. Two possibilities are memory mapping of data structures into the kernel address space, which cannot be detected by examination of pointers in user address space, and the use of only interior pointers to data.

To solve this problem, we added the *alien bit* to

the header. If *alien* is set, the segment is not freed even if it remains unmarked during garbage collection. We force all allocated segments to be set as *aliens* during early initialization, and once the dangerous allocations are done, return to ordinary operation. This seems to correct the problem. In our program, only a few thousand bytes in ten or so segments are so treated.

Valloc

Another surprise was caused by the *valloc* library call. *valloc* is a variant of *malloc* that allocates space on a page boundary; it is called by the Sun graphics code. We had believed that this call was layered on top of the standard allocator, but on further investigation, learned that it had knowledge of the specific header format and free list layout used by the allocator. To function, *valloc* allocates a block large enough to contain one memory page, then frees the excess at the beginning of the segment by adjusting its header and creating a new header in the aligned section. (We deduced this from Sun documentation; the VAX version of *valloc* does not behave this way, but the pointer it returns is an interior pointer, and so cannot be freed by *free*.)

We implemented *valloc* correctly by integrating it into our memory allocator and setting segment headers appropriately.

Realloc

After our program began to function correctly with the new allocator, we were surprised by very long pauses during certain operations — much longer than with the standard allocator. We tracked this down to the fact that the *realloc* function was being heavily used by those portions of the code.

The call

```
char *p; /* previously set to point
          at a size P segment */
```

```
p = realloc(p, N);
```

is an optimized version of the sequence

```
char *p;
char *p2;

p2 = malloc(N);
bcopy(p, p2, min(P, N));
free(p);
p = p2;
```

If *N* is greater than *P*, *realloc* tries to grow the segment in place and avoid the copy; otherwise, it shrinks the segment, creating a new free one at its end.

The slowly-executing program code was formatting a character string line-by-line and appending each new line to a result string with *realloc*. We had implemented *realloc* in the naive way, so the CPU time was going into copying and allocation (and also fragmenting the heap, as more and more small regions

were allocated and then immediately discarded.) Note that since the 4.2 *malloc* tends to leave dead space at the end of each segment, it can often perform expanding *reallocs* cheaply.

We solved this problem by making our *realloc* clever enough to do the same trick. Once done, our allocator became competitive with the standard one.

Performance

Allocation

For evaluation purposes, we instrumented the Telesophy interface and generated a trace of all memory operations performed during a long session. The activity consisted of 15815 operations, broken down as follows:

malloc	1769
realloc	13684
free	362

It's immediately obvious why an inefficient *realloc* has such an impact on this program.

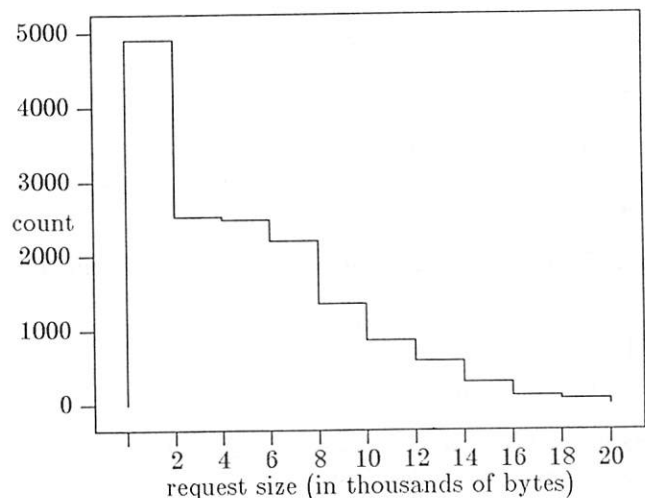


Figure 2: Histogram of Request Sizes

Figure 2 shows a histogram of the sizes of requests for both *malloc* and *realloc*. Most of the allocations in the middle of the range are *reallocs*; nearly all of the *malloc* requests were for blocks less than 500 bytes in size. The total amount of space allocated by both *malloc* and *realloc* is about 75 megabytes; little more than a megabyte is allocated by *malloc* directly.

A test program, executing just the operations in the trace, used the CPU time show in Figure 3⁴.

The first two times are from the new Sun *malloc* and the 4.2 version described earlier. Note that the latter is considerably faster than either Sun's or ours, indicating that Sun is now using a different allocation policy (or perhaps doing more checking.) Our allocator took the most user time to run through the trace,

⁴Timings are in CPU seconds on a Sun-3/75 running release 3.2 of Sun Unix.

probably because it has to scan more segments during allocation. In a more *malloc*-intensive trace, ours might fall further behind. It is interesting to note that our version used the least system time. We suspect this is because we request larger expansions at a time via *sbrk* than the others do, cutting down on the number of system calls.

allocator	user time	system time
Sun	10.1	0.7
4.2	7.7	1.0
GC	10.3	0.3

Figure 3

Note that no garbage collections were actually performed by our allocator; the purpose of these timings was to verify that we were not paying an excessive penalty to use an allocator that also garbage collected. We feel that goal was attained.

Garbage collection

We modified the test program so that it never performed any frees, and put a soft limit on memory of 512K bytes, then ran it on the trace used before. The total CPU time increased considerably, to 51 user seconds, of which about 38 seconds was consumed by the nine invocations of the garbage collector – including one collection for 6 cpu seconds. It should be pointed out that the behavior of the test program is somewhat unrealistic; it uses a single global array of pointers to store the results of allocation. In the program that actually generates the trace, many pointers are local and become inaccessible when their blocks are exited, and garbage collection is much more effective.

Garbage collection performance is not extremely good, but it is acceptable. We put the same limits (no freeing, 512K bytes) on the actual Telesophy front end; though it did GC too often for really comfortable interactive use, it did correctly function in 512K bytes of memory.

More importantly, when we installed the new allocator in our server programs (which are idle for long periods between accesses and can GC without concern for interactive response) we no longer worried about consuming too much virtual memory. In addition, a data-importing program that had been crashing due to memory exhaustion after processing three weeks of *New York Times* data successfully processed four months' worth to completion, after the garbage collector was added. We happily traded the minute or so of CPU time of the garbage collects for the hours of programmer time that would have been needed to track down storage leaks.

Conclusions

We have described a memory allocator package, completely compatible with the existing Unix *malloc* calls, that performs automatic garbage collection in C

programs. So far, we are extremely pleased with its functionality, though performance could be better for interactive programs.

The allocator consists of about 600 lines of C and a few dozen lines of machine-specific assembler. Versions have been written for the VAX and 680x0-based Sun machines; we anticipate posting this software to the Usenet shortly.

The presence of the garbage collector has made programming quite a bit easier in our applications, and we hope that the release of this allocator and its porting to machines other than the Sun and VAX will make memory allocation problems more manageable for the C programming community.

Acknowledgements

The conservative approach was first suggested to me by David Chase at Rice University, though neither of us believed it was really feasible at the time. Discussions with Hans Boehm, also of Rice, who had implemented a similar (but not *malloc*-compatible) system for his Russell compiler, convinced me the approach would work. That system is described in [Boehm], and has been used as part of the Portable Cedar system by Mark Weiser at Xerox PARC. Hans also provided the register-examining assembly language and pointed out a redundancy in a previous garbage collection algorithm. Mike Bianchi of Bellcore reviewed the paper and suggested a number of possible enhancements to the collector, including the following-block optimization for *free*. I also thank Joel Gannett of Bellcore and John Gilmore for their extensive comments.

References

- Boehm. Hans-Juergen Boehm, "Garbage Collection in an Uncooperative Environment," submitted to *Software—Practice and Experience*, Rice University (1987).
- Caplinger. Michael Caplinger, "An Information System using Distributed Objects," *OOPSLA '87 Conference Proceedings*, (to appear October 1987).
- Christopher. Thomas Christopher, "Reference Count Garbage Collection," *Software—Practice and Experience* 14(6) pp. 503-507 (June 1984).
- Knuth. Don Knuth, "Section 2.3.5, Lists and Garbage Collection," in *The Art of Computer Programming, Volume 1 (Fundamental Algorithms)*, Addison-Wesley (1969).
- Korn. David Korn and Kiem-Phong Vo, "In Search of a Better Malloc," *Proceedings of the 1985 Summer Usenix Conference*, pp. 489-506 (1985).
- Rovner. Paul Rovner, "On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-checked, Concurrent Language," CSL-84-7, Xerox Palo Alto Research Center (1984).



Rescuing Data in UNIX File Systems (What to do after rm *)

Jim Joyce
Bob Nystrom
The Gawain Group
47 Potomac St
SF, CA 94117
415/626-7581

ABSTRACT

You have just typed
`rm precious. *`
and pressed return. You receive the message
`precious.: No such file or directory`
and have now seen the space between the period and the asterisk. Now what? The backups weren't made recently enough, and besides they don't include today's changes.

It is unfortunately common to make this type of error, removing or truncating precious files. This paper will explore methods for recovering inadvertently removed data files with as minimal data loss as possible.

This paper presents algorithms for lost data retrieval that have been used to recover data successfully from clobbered files and file systems. These algorithms have resulted in data recovery effectiveness in excess of 99% on AT&T versions of UNIX.

The paper discusses, as a future development, a more global algorithm for data recovery. Given a readable disk, the algorithm will find the file systems present on the drive and automatically assign likelihood values regarding the type of data contained in each block. This information may then be used to recover data from a badly clobbered file system or from a disk partitioned for a foreign system.

Lastly, we offer a checklist for file recovery, setting forth steps to follow in assessing an apparent problem.

`cat > precious`

Our Data Rescue Service came about as a response to a client downloading over \$15,000 in billing data from one computer system to a UNIX system. The transfer was controlled by a shell script, and the key command was

```
cat > precious
```

where `precious`[†] was to hold the billing data until the UNIX system could process the bills. A data transfer was in progress when the client suddenly felt something was wrong, and stopped the transfer.

After being reassured that all was well, the transfer was restarted - only to find that the other computer system had truncated its billing file and was transferring only the few records that had accumulated since the original transfer had taken place. The client pulled the UNIX system's power cord out of the wall socket to prevent further problems and, after a

few referrals, called us. Though pulling the power cord out of the wall socket is a rather extreme measure, this is perhaps one of the few instances when it has been the right thing to do in response to a problem.

Basic structure of a UNIX file system

Those who already understand the basic structure of a UNIX file system should skip to the next section.

A UNIX file system is composed of five major structures:

- * Boot Block
- * Super Block
- * I-list
- * Data Blocks

The Boot Block used to contain the bootstrap program for booting UNIX, but no longer does.

The Super Block is the major Table of Contents for the file system. Its structure is specified in `/usr/include/sys/filsys.h` - though some

[†]The story you are reading is true. The names have been changed to prevent undue embarrassment.

versions of UNIX/XENIX hide it elsewhere. Although all of the fields in the Super Block are important,

Table 1	
Name	Contents
<code>s_isize</code>	size of i-list
<code>s_fsize</code>	size of entire volume
<code>s_nfree</code>	no. of addresses in <code>s_free</code>
<code>s_free[NICFREE]</code>	free block list
<code>s_ninode</code>	no. of i-nodes in <code>s_inode</code>
<code>s_inode[NICINOD]</code>	free i-node list
<code>s_tfree</code>	total free blocks
<code>s_tinode</code>	total free inodes
<code>s_magic</code>	magic no. for file system

those of most interest to us here are in Table 1.

The i-list is a sequence of i-nodes, or information nodes. These are structures containing data about an individual file. For our purposes the most useful data in an i-node are the thirteen pointers to data blocks. The first ten are pointers directly to data; the eleventh is a pointer to an indirect block of 128 pointers to blocks of data; the twelfth pointer is to a double-indirect block, containing 128 pointers to indirect blocks; and the thirteenth pointer is to a triple-indirect block, containing 128 pointers to indirect blocks. All of these pointers define the order of blocks in a file.

The data blocks make up the remainder of the UNIX file system. Data blocks may contain user data, but they also contain the indirect blocks pointed to by the i-node, and they also contain directories.

Notably, the Super Block contains information about free i-nodes and free blocks; allocated (or owned) blocks are accounted for in the i-nodes. If a block is not pointed to by an i-node, however indirectly, the `fsck` utility will consider it to be a free block and put it on the Free List.

What happens when a file is deleted

Readers who already know that `s_free[NICFREE]` is copied to the block pointed to by `s_free[0]` may skip this section.

A file may be deleted by a user on System V in two basic ways:

```
rm precious
```

and

```
cat > precious
```

— and though other commands may also destroy existing data they do so either by removing or overwriting. In the case of `rm precious`, blocks are returned to

the Free List, and the i-node is put on `s_inode[NICINOD]` in the Super Block. In the case of `cat > precious`, blocks are returned to the Free List, and all i-node extent pointers are erased because the i-node is re-used.

When blocks are returned to the Free List, they are placed on the *FRONT* of the Free List. They are deallocated as a FIFO to reduce fragmentation. The block chain is recorded in the Super Block in `s_free[NICFREE]`. And now, the bad news: the chain is also copied to the block whose number is in `s_free[0]`.

Such copying does destroy some of the data, making 100% recovery impossible under System V. But the good news is that the data in the rest of the block is untouched. Blocks on the Free List are reallocated from the *end* of `s_free[NICFREE]`. So, if data is to be rescued from the Free List, it had better be done before other files are created on that file system.

In the case of our Data Rescue client, there was no other activity on the system at the time of the disaster. This was auspicious in that it suggested the billing data might be fairly intact on the Free List. Had there been much activity on the system the chances of recovering precious would have been much less.

Make a copy of the file system

We agreed to investigate just how much of the billing data we might be able to recover, and instructed our client to mount the disk read-only as a second hard disk on an identical system and then to create a cartridge tape copy of the entire file system using

```
dd if=/dev/rhd1a of=/dev/rct obs=5120
```

where `rhd1a` is the name of the disk partition where we hoped the data still was, and `rct` is the name of the cartridge tape drive. We used the raw device name to skip kernel buffering of the data in addition to `dd`'s specified buffering.

Verify that the copy is readable

After having the client verify the cartridge tape was readable using

```
dd if=/dev/rct of=/dev/null bs=5120
```

we suggested that the disk be unmounted, unplugged and kept in a safe place until we could report on our progress — or lack of it. In this way we were able to get access to a copy of the entire disk without putting the disk itself at risk by having it shipped to us.

Using system data structures

Since timeliness was important, as we awaited the arrival of the tape we began planning. We would transfer the tape's contents to a single disk file. We designed a suite of programs to read the file, examining its various blocks using system data structures to locate the Free List. And, hopefully, the billing data

our client wanted. The system data structures in `/usr/include/sys` were the obvious platform to use in recognizing whether a data block was

- * a Super Block
- * an i-node
- * a directory
- * an indirect block
- * a double-indirect block
- * a triple-indirect block
- * a data block

We also asked our client to send a sample layout of the data to be retrieved.

Designing the data rescue programs

We brainstormed possible problems in recognizing what type a given data block might be. We concluded that in the worst case every block could be every kind of block. We fervently hope never to find that situation in an actual data rescue attempt. Still, our analysis argued for generality in identifying blocks. We designed data structures for typing blocks to accommodate likelihood values for a given block being a certain type. These likelihood values are *fuzzy values*.

Fuzzy values [Zadeh65] are values between 0 and 1 that indicate the *degree* to which an element is a member of a set. They are determined by a *membership function* for the set in question. Each type of data block would have its own membership function, and each block would be measured for membership in each type of data block.

So, a data block with a Super Block fuzzy value of 0 is *not* a Super Block, whereas one with a fuzzy value of 1 means it *is* a Super Block. The reality of damaged file systems is that a Super Block might be only partially intact, and rate a fuzzy value somewhere between 0 and 1. Currently deemed sufficient for identifying a Super Block is that if `s_magic` contains the value `FSMAGIC`, as defined in `filsys.h`, we say we have found a Super Block.

Constructing the membership functions is only partially complete at present because we needed first and foremost to rescue our client's billing data. We also intend to improve upon the simple reliance on `s_magic`'s value.

In System V the Boot Block is supposed to be block 0, and the Super Block is supposed to be block 1. However, we chose to design for the general case, in which any of the blocks was a candidate for being a Super Block. The Berkeley Fast File System and some versions of Altos XENIX have more than one copy of the Super Block. And if block 1 were trashed we would be more than happy to use another copy of the Super Block if we could find it.

We designed `scandisk.c`, which performs an initial scan of the disk image trying to make a best-guess as to the content of each disk block. One of the routines it calls is `isfilsys.c`, which contains routines for processing a Super Block. With

`isfilsys.c`, we initialize appropriate data structures. We must remember to be leery about simply trusting the Super Block entries, as they may be incorrect. However, we use the information reported if we can. Once there is some notion of the makeup of the file system we proceed with `isinode.c`, processing candidate i-node entries. Again, the Super Block information is used as a guide, if we can.

Getting to work

Once with the tape in our possession, and using a system lent to us by Altos Computer Systems that could read the cartridge tape, we set about reading the dd image onto disk. Then we began examination of our file containing Super Block, i-list, and data blocks.

Things went amazingly close to our planned rescue strategy. We first determined the size and number of the disk blocks. Then we found the Super Block, and used the information there to initialize data structures that would aid in tracing through the Free List. We were prepared to do without valid Super Block data since the machine had been shut down so abruptly. This time our client lucked out: we found valid Super Block data.

Because

```
cat > precious
```

directed output to an existing file we felt sure that the i-node was scrubbed of the old information in preparation for storing information about the new file. Depending upon just when the power was cut, i-node processing might not have been completed and the i-node pointers might still be there. So, just to be sure we did not overlook anything that might be helpful we examined the i-node; sure enough, the pointers to the previous file's data blocks had been erased. We faced the prospect of tracing through the disk image, looking at blocks and evaluating what kind they were.

Our routine `initflist.c` walks through the Free List and grabs the blocks from it in the order they were placed there. The result is a best approximation of the file as it would have been before being removed.

Because the Free List obviously contains more than one deleted file, we retrieved more than the desired billing file. This embarrassment of riches was easily resolved through shell tools, since we had a sample of the data format. Armed with the Free List block numbers, and `flcat.c` to copy blocks, we used a 14-line shell script to display the block using `od`, asking whether we wanted it saved, and then appending the block to its fellow data blocks.

Because `s_free[0]` had a copy of `s_free[NICFREE]` in it, there was destroyed data. We translated those bytes to `@` to avoid confusion with valid data. Though some government agencies are rumored to be able to read data in blocks that have been overwritten, we do not have that technology available to us.

Finally, we made a tape of the rescued file and sent it back. They were very happy to have the data back.

Sometimes it IS hardware

Our software for data recovery supposes readable disk, diskette, or magnetic tape. But what if the media cannot be read? All is not necessarily lost. Three instances occurred in which the media at first refused to be read but later yielded data.

The system on which this paper is being written suffered a disk failure – the drive signaled it was not ready. Backups were recent enough, but the prospect of trying to recover the disk was irresistible. Was the problem truly a disk failure or only an apparent failure.

The steps were to power down, check that cable connections were snug, and clean the fan filters. After disconnecting and removing the disk, we then blew air using an ordinary hair dryer to dispel dust that might have lodged in the disk electronics.

After reconnecting everything the system powered up, `fsck` deleted a zero-length file, and the system has been running ever since. The advent of supermicros in the work environment rather than in special rooms makes this data recovery memorable. Blowing out dust in a system and cleaning the filters should be a scheduled activity, just like doing backups.

The second media problem involved a tape backup of hospital billing files that was marred by bad spots. The solution was to `dd` the tape to disk with `conv=noerror` and recover the files `tar` could read. Then, armed with another backup tape from a day or so earlier, recover the earlier version of the files. Though the recovery was time-consuming, having two backup tapes made it possible to do. Of course, verifying the tape shortly after creation would have revealed the bad spots before the matter became a data rescue item.

Third in the media recovery instances is one in which the electronics on a disk drive was replaced successfully. A cold solder joint was the culprit, which led to a large recall of this particular disk drive from a European shipment of systems. The drive itself was good, and the data intact; but the electronics cabled to the drive was defective.

Limitations, and what's next?

Though we have designed with generality and flexibility as key principles, the software works for systems with a free block list. Berkeley-based systems, with a bit-mapped free list, are the next target for data rescue software. In conversation with Kirk McKusick one of the authors confirmed that deleted blocks in a Berkeley system are not scribbled on as they are in System V, and so in theory 100% data recovery is possible.

A more serious limitation is that of attempted recovery of a program that has been deleted. Missing parts in data files are one thing, but missing parts of a program appear to doom data recovery efforts at the outset. Our clients would appreciate any suggestions.

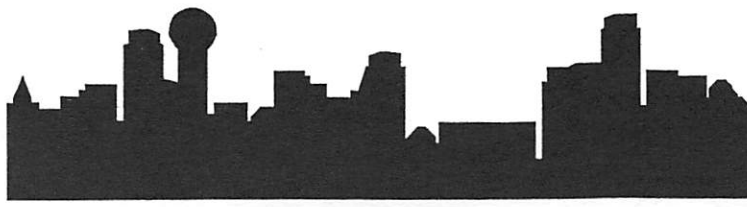
Checklist for file recovery

We close with a Checklist for file recovery. It provides basic guidance only.

1. Don't panic()
Is it *really* lost?
+ You may not really be in trouble yet, but if you panic you just might do something rash that does real damage.
+ If it is lost, try step 2.
2. Calmly determine the worth of your lost data.
+ Is it really *precious*?
+ What is the cost of re-entering the data?
+ What is the cost of recovering the data?
+ should you shut down the system?
3. Stop system activity ASAP.
+ If only user, shut down.
+ Pull the plug *may* be expedient.
+ Delete a large file to provide a cushion to *pre-* cious on the Free List, but only if *it* is backed up.
+ Do not wait – activity is the enemy.
4. Do NOT risk the data.
+ Mount file system read only
+ `dd` disk to back up medium.
+ May need `conv=noerror` option to skip over read errors.
+ Keep original safe until prognosis/results.
5. Plan your recovery action.
+ Is the problem hardware or software?
+ Do it yourself?
+ Call us?

References

- [Zadeh65] L. A. Zadeh, "Fuzzy Sets," *Inf. and Cont.*, 8, 338-353 (1965).



How To Steal Code -or- Inventing The Wheel Only Once

Henry Spencer
Zoology Computer Systems
25 Harbord St.
University of Toronto
Toronto, Ont. M5S 1A1 Canada
{allegra,ihnp4,decvax,utai}@utzo.henry

ABSTRACT

Much is said about "standing on other people's shoulders, not their toes", but in fact the wheel is re-invented every day in the UNIX/C community. Worse, often it is re-invented badly, with bumps, corners, and cracks. There are ways of avoiding this: some of them bad, some of them good, most of them under-appreciated and under-used.

Introduction

"Everyone knows" that that the UNIX/C community and its programmers are the very paragons of re-use of software. In some ways this is true. Brian Kernighan [1] and others have waxed eloquent about how outstanding UNIX is as an environment for software re-use. Pipes, the shell, and the design of programs as 'filters' do much to encourage programmers to build on others' work rather than starting from scratch. Major applications can be, and often are, written without a line of C. Of course, there are always people who insist on doing everything themselves, often citing 'efficiency' as the compelling reason why they can't possibly build on the work of others (see [2] for some commentary on this). But surely these are the lamentable exceptions, rather than the rule?

Well, in a word, no.

At the level of shell programming, yes, software re-use is widespread in the UNIX/C community. Not quite as widespread or as effective as it might be, but definitely common. When the time comes to write programs in C, however, the situation changes. It took a radical change in directory format to make people use a library to read directories. Many new programs still contain hand-crafted code to analyze their arguments, even though prefabricated help for this has been available for years. C programmers tend to think that "re-using software" means being able to take the source for an existing program and edit it to produce the source for a new one. While that *is* a useful technique, there are better ways.

Why does it matter that re-invention is rampant? Apart from the obvious, that programmers have more work to do, I mean? Well, extra work for the programmers is not exactly an unmixed blessing, even from the programmers' viewpoint! Time spent re-inventing facilities that are already available is

time that is *not* available to improve user interfaces, or to make the program run faster, or to chase down the proverbial Last Bug. Or, to get really picky, to make the code readable and clear so that our successors can *understand* it.

Even more seriously, re-invented wheels are often square. Every time that a line of code is re-typed is a new chance for bugs to be introduced. There will always be the temptation to take shortcuts based on how the code will be used—shortcuts that may turn around and bite the programmer when the program is modified or used for something unexpected. An inferior algorithm may be used because it's "good enough" and the better algorithms are too difficult to reproduce on the spur of the moment... but the definition of "good enough" may change later. And unless the program is well-commented [here we pause for laughter], the next person who works on it will have to study the code at length to dispel the suspicion that there is some subtle reason for the seeming re-invention. Finally, to quote [2], *if you re-invent the square wheel, you will not benefit when somebody else rounds off the corners.*

In short, re-inventing the wheel ought to be a rare event, occurring only for the most compelling reasons. Using an existing wheel, or improving an existing one, is usually superior in a variety of ways. There is nothing dishonorable about stealing code¹ to make life easier and better.

Theft via the Editor

UNIX historically has flourished in environments in which full sources for the system are available. This led to the most obvious and crudest way of stealing code: copy the source of an existing program and edit it to do something new.

¹Assuming no software licenses, copyrights, patents, etc. are violated!

This approach does have its advantages. By its nature, it is the most flexible method of stealing code. It may be the only viable approach when what is desired is some variant of a complex algorithm that exists only within an existing program; a good example was V7 *dumpdir* (which printed a table of contents of a backup tape), visibly a modified copy of V7 *restor* (the only other program that understood the obscure format of backup tapes). And it certainly is easy.

On the other hand, this approach also has its problems. It creates two subtly-different copies of the same code, which have to be maintained separately. Worse, they often have to be maintained “separately but simultaneously”, because the new program inherits all the mistakes of the original. Fixing the same bug repeatedly is so mind-deadening that there is great temptation to fix it in only the program that is actually giving trouble... which means that when the other gives trouble, re-doing the cure must be preceded by re-doing the investigation and diagnosis. Still worse, such non-simultaneous bug fixes cause the variants of the code to diverge steadily. This is also true of improvements and cleanup work.

A program created in this way may also be inferior, in some ways, to one created from scratch. Often there will be vestigial code left over from the program’s evolutionary ancestors. Apart from consuming resources (and possibly harboring bugs) without a useful purpose, such vestigial code greatly complicates understanding the new program in isolation.

There is also the possibility that the new program has inherited a poor algorithm from the old one. This is actually a universal problem with stealing code, but it is especially troublesome with this technique because the original program probably was not built with such re-use in mind. Even if its algorithms were good for its intended purpose, they may not be versatile enough to do a good job in their new role.

One relatively clean form of theft via editing is to alter the original program’s source to generate either desired program by conditional compilation. This eliminates most of the problems. Unfortunately, it does so only if the two programs are sufficiently similar that they can share most of the source. When they diverge significantly, the result can be a maintenance nightmare, actually worse than two separate sources. Given a close similarity, though, this method can work well.

Theft via Libraries

The obvious way of using somebody else’s code is to call a library function. Here, UNIX has had some success stories. Almost everybody uses the *stdio* library rather than inventing their own buffered-I/O package. (That may sound trivial to those who never programmed on a V6 or earlier UNIX, but in fact it’s a great improvement on the earlier state of affairs.) The simpler sorts of string manipulations are usually

done with the *strxxx* functions rather than by hand-coding them, although efficiency issues and the wide diversity of requirements have limited these functions to less complete success. Nobody who knows about *qsort* bothers to write his own sorting function.

However, these success stories are pleasant islands in an ocean of mud. The fact is that UNIX’s libraries are a disgrace. They are well enough implemented, and their design flaws are seldom more than nuisances, but there aren’t *enough* of them! Ironically, UNIX’s “poor cousin”, the Software Tools community [3,4], has done much better at this. Faced with a wild diversity of different operating systems, they were forced to put much more emphasis on identifying clean abstractions for system services.

For example, the Software Tools version of *ls* runs unchanged, *without* conditional compilation, on dozens of different operating systems [4]. By contrast, UNIX programs that read directories invariably dealt with the raw system data structures, until Berkeley turned this cozy little world upside-down with a change to those data structures. The Berkeley implementors were wise enough to provide a library for directory access, rather than just documenting the new underlying structure. However, true to the UNIX pattern, they designed a library which quietly assumed (in some of its naming conventions) that the underlying system used *their* structures! This particular nettle has finally been grasped firmly by the IEEE POSIX project [5], at the cost of yet another slightly-incompatible interface.

The adoption of the new directory libraries is not just a matter of convenience and portability: in general the libraries are faster than the hand-cooked code they replace. Nevertheless, Berkeley’s original announcement of the change was greeted with a storm of outraged protest.

Directories, alas, are not an isolated example. The UNIX/C community simply hasn’t made much of an effort to identify common code and package it for re-use. One of the two major variants of UNIX still lacks a library function for binary search, an algorithm which is notorious for both the performance boost it can produce and the difficulty of coding a fully-correct version from scratch. No major variant of UNIX has a library function for either one of the following code fragments, both omnipresent (or at least, they *should* be omnipresent [6]) in simple² programs that use the relevant facilities:

```
if ((f = fopen(filename, mode))
    == NULL)
    print error message with
      filename, mode, and specific
      reason for failure, and then exit
```

²I include the qualification “simple” because complex programs often want to do more intelligent error recovery than these code fragments suggest. However, *most* of the programs that use these functions *don’t* need fancy error recovery, and the error responses indicated are *better* than the ones those programs usually have now!

```
if ((p = malloc(amount)) == NULL)
    print error message and exit
```

These may sound utterly trivial, but in fact programmers almost never produce as good an error message for *fopen* as ten lines of library code can, and half the time the return value from *malloc* isn't checked at all!

These examples illustrate a general principle, a side benefit of stealing code: the way to encourage standardization³ and quality is to make it easier to be careful and standard than to be sloppy and non-standard. On systems with library functions for error-checked *fopen* and *malloc*, it is easier to use the system functions—which take some care to do “the right thing”—than to kludge it yourself. This makes converts very quickly.

These are not isolated examples. Studying the libraries of most any UNIXnon- system will yield other ideas for useful library functions (as well as a lot of silly nonsense that UNIX doesn't need, usually!). A few years of UNIX systems programming also leads to recognition of repeated needs. Does *your*⁴ UNIX have library functions to:

- decide whether a filename is well-formed (contains no control characters, shell metacharacters, or white space, and is within any name-length limits your system sets)?
- close all file descriptors except the standard ones?
- compute a standard CRC (Cyclic Redundancy Check “checksum”)?
- operate on *malloced* unlimited-length strings?
- do what *access(2)* does but using the effective *userid*?
- expand metacharacters in a filename the same way the shell does? (the simplest way to make sure that the two agree is to use *popen* and *echo* for anything complicated)
- convert integer baud rates to and from the speed codes used by your system's serial-line *ioctl*s?
- convert integer file modes to and from the *rw*x strings used⁵ to present such modes to humans?
- do a binary search through a file the way *look(1)* does?

The above are fairly trivial examples of the sort of things that *ought* to be in UNIX libraries. More sophisticated libraries can also be useful, especially if the language provides better support for them than C

³Speaking of encouraging standardization: we use the names *efopen* and *emalloc* for the checked versions of *fopen* and *malloc*, and arguments and returned values are the same as the unchecked versions except that the returned value is guaranteed non-NULL if the function returns at all.

⁴As you might guess, my system has all of these. Most of them are trivial to write, or are available in public-domain forms.

⁵If you think only *ls* uses these, consider that *rm* and some similar programs *ought* to use *rw*x strings, not octal modes, when requesting confirmation!

does; C++ is an example [7]. Even in C, though, there is much room for improvement.

Adding library functions does have its disadvantages. The interface to a library function is important, and getting it right is hard. Worse, once users have started using one version of an interface, changing it is very difficult even when hindsight clearly shows mistakes; the near-useless return values of some of the common UNIX library functions are obvious examples. Satisfactory handling of error conditions can be difficult. (For example, the error-checking *malloc* mentioned earlier is very handy for programmers, but invoking it from a library function would be a serious mistake, removing any possibility of more intelligent response to that error.) And there is the perennial headache of trying to get others to adopt your pet function, so that programs using it can be portable without having to drag the source of the function around too. For all this, though, libraries are in many ways the most satisfactory way of encouraging code theft.

Alas, encouraging code theft does not guarantee it. Even widely-available library functions often are not used nearly as much as they should be. A conspicuous example is *getopt*, for command-line argument parsing. *Getopt* supplies only quite modest help in parsing the command line, but the standardization and consistency that its use produces is still quite valuable; there are far too many pointless variations in command syntax in the hand-cooked argument parsers in most UNIX programs. Public-domain implementations of *getopt* have been available for years, and AT&T has published (!) the source for the System V implementation. Yet people continue to write their own argument parsers. There is one valid reason for this, to be discussed in the next section. There are also a number of excuses, mostly the standard ones for not using library functions:

- “It doesn't do quite what I want.” *But often it is close enough to serve, and the combined benefits of code theft and standardization outweigh the minor mismatches.*
- “Calling a library function is too inefficient.” *This is mostly heard from people who have never profiled their programs and hence have no reliable information about what their code's efficiency problems are [2].*
- “I didn't know about it.” *Competent programmers know the contents of their toolboxes.*
- “That whole concept is ugly, and should be redesigned.” (Often said of *getopt*, since the usual UNIX single-letter-option syntax that *getopt* implements is widely criticized as user-hostile.) *How likely is it that the rest of the world will go along with your redesign (assuming you ever finish it)? Consistency and a high-quality implementation are valuable even if the standard being implemented is suboptimal.*

“I would have done it differently.” *The triumph*

of personal taste over professional programming.

Theft via Templates

Templates are a major and much-neglected approach to code sharing: “boilerplate” programs which contain a carefully-written skeleton for some moderately stereotyped task, which can then be adapted and filled in as needed. This method has some of the vices of modifying existing programs, but the template can be designed for the purpose, with attention to quality and versatility.

Templates can be particularly useful when library functions are used in a stereotyped way that is a little complicated to write from scratch; *getopt* is an excellent example. The one really valid objection to *getopt* is that its invocation is not trivial, and typing in the correct sequence from scratch is a real test of memory. The usual *getopt* manual page contains a lengthy example which is essentially a template for a *getopt*-using program.

When the first public-domain *getopt* appeared, it quickly became clear that it would be convenient to have a template for its use handy. This template eventually grew to incorporate a number of other things: a useful macro or two, definition of *main*, opening of files in the standard UNIX filter fashion, checking for mistakes like opening a directory, filename and line-number tracking for error messages, and some odds and ends. The full current version can be found in the Appendix; actually it diverged into two distinct versions when it became clear that some filters wanted the illusion of a single input stream, while others wanted to handle each input file individually (or didn't care).

The obvious objection to this line of development is “it's more complicated than I need”. In fact, it turns out to be surprisingly convenient to have all this machinery presupplied. *It is much easier to alter or delete lines of code than to add them.* If directories are legitimate input, just delete the code that catches them. If no filenames are allowed as input, or exactly one must be present, change one line of code to enforce the restriction and a few more to deal with the arguments correctly. If the arguments are not filenames at all, just delete the bits of code that assume they are. And so forth.

The job of writing an ordinary filter-like program is reduced to filling in two or three blanks⁶ in the template, and then writing the code that actually processes the data. Even quick improvisations become good-quality programs, doing things the standard way with all the proper amenities, because even a quick improvisation is easier to do by starting from the template. *Templates are an unmixed blessing; anyone who types a non-trivial program in from scratch is wasting his time and his employer's money.*

Templates are also useful for other stereotyped

files, even ones that are not usually thought of as programs. Most versions of UNIX have a simple template for manual pages hiding somewhere (in V7 it was */usr/man/man0/xx*). Shell files that want to analyze complex argument lists have the same *getopt* problem as C programs, with the same solution. There is enough machinery in a “production-grade” *make* file to make a template worthwhile, although this one tends to get altered fairly heavily; our current one is in the Appendix.

Theft via Inclusion

Source inclusion (*#include*) provides a way of sharing both data structures and executable code. Header files (e.g. *stdio.h*) in particular tend to be taken for granted. Again, those who haven't been around long enough to remember V6 UNIX may have trouble grasping what a revolution it was when V7 introduced systematic use of header files!

However, even mundane header files could be rather more useful than they normally are now. Data structures in header files are widely accepted, but there is somewhat less use of them to declare the return types of functions. One or two common header files like *stdio.h* and *math.h* do this, but programmers are still used to the idea that the type of (e.g.) *atol* has to be typed in by hand. Actually, all too often the programmer says “oh well, on my machine it works out all right if I don't bother declaring *atol*”, and the result is dirty and unportable code. The X3J11 draft ANSI standard for C addresses this by defining some more header files and requiring their use for portable programs, so that the header files can do all the work and do it *right*.

In principle, source inclusion can be used for more than just header files. In practice, almost anything that can be done with source inclusion can be done, and usually done more cleanly, with header files and libraries. There are occasional specialized exceptions, such as using macro definitions and source inclusion to fake parameterized data types.

Theft via Invocation

Finally, it is often possible to steal another program's code simply by invoking that program. Invoking other programs via *system* or *popen* for things that are easily done in C is a common beginner's error. More experienced programmers can go too far the other way, however, insisting on doing everything in C, even when a leavening of other methods would give better results. The best way to sort a large file is probably to invoke *sort(1)*, not to do it yourself. Even invoking a shell file can be useful, although a bit odd-seeming to most C programmers, when elaborate file manipulation is needed and efficiency is not critical.

Aside from invoking other programs at run time, it can also be useful to invoke them at compile time. Particularly when dealing with large tables, it is often

⁶All marked with the string ‘xxx’ to make them easy for a text editor to find.

better to dynamically generate the C code from some more compact and readable notation. *Yacc* and *lex* are familiar examples of this on a large scale, but simple *sed* and *awk* programs can build tables in more specialized, application-specific ways. Whether this is really theft is debatable, but it's a valuable technique all the same. It can neatly bypass a lot of objections that start with "but C won't let me write..."

An Excess of Invention

With all these varied methods, why is code theft not more widespread? Why are so many programs unnecessarily invented from scratch?

The most obvious answer is the hardest to counter: theft requires that there be something to steal. Use of library functions is impossible unless somebody sets up a library. Designing the interfaces for library functions is not easy. Worse, doing it *well* requires insight, which generally isn't available on demand. The same is true, to varying degrees, for the other forms of theft.

Despite its reputation as a hotbed of software re-use, UNIX is actually hostile to some of these activities. If UNIX directories had been complex and obscure, directory-reading libraries would have been present from the beginning. As it is, it was simply *too easy* to do things "the hard way". There *still* is no portable set of functions to perform the dozen or so useful manipulations of terminal modes that a user program might want to do, a major nuisance because changing those modes "in the raw" is simple but highly unportable.

Finally, there is the Not Invented Here syndrome, and its relatives, Not Good Enough and Not Understood Here. How else to explain AT&T UNIX's persistent lack of the *dbm* library for hashed databases (even though it was developed at Bell Labs and hence is available to AT&T), and Berkeley UNIX's persistent lack of the full set of *strxxx* functions (even though a public-domain implementation has existed for years)? The X3J11 and POSIX efforts are making some progress at developing a common nucleus of functionality, but they are aiming at a common subset of current systems, when what is really wanted is a common superset.

Conclusion

In short, never build what you can (legally) steal! Done right, it yields better programs for less work.

References

- [1] Brian W. Kernighan, *The Unix System and Software Reusability*, IEEE Transactions on Software Engineering, Vol SE-10, No. 5, Sept. 1984, pp. 513-8.
- [2] Geoff Collyer and Henry Spencer, *News Need Not Be Slow*, Usenix Winter 1987 Technical Conference, pp. 181-190.
- [3] Brian W. Kernighan and P.J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass. 1976.
- [4] Mike O'Dell, *UNIX: The World View*, Usenix Winter 1987 Technical Conference, pp. 35-45.
- [5] IEEE, *IEEE Trial-Use Standard 1003.1 (April 1986): Portable Operating System for Computer Environments*, IEEE and Wiley-Interscience, New York, 1986.
- [6] Ian Darwin and Geoff Collyer, *Can't Happen or /* NOTREACHED */ or Real Programs Dump Core*, Usenix Winter 1985 Technical Conference, pp. 136-151.
- [7] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass. 1986.

Appendix

Warning: these templates have been in use for varying lengths of time, and are not necessarily all entirely bug-free.

C program, single stream of input

```
/*
 * name - purpose xxx
 *
 * $Log$
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

#define MAXSTR 500          /* For sizing strings -- DON'T use BUFSIZ! */
#define STREQ(a, b)        ((a) == (b) && strcmp((a), (b)) == 0)

#ifndef lint
static char RCSid[] = "$Header$";
#endif

int debug = 0;
char *progname;

char **argvp;                /* scan pointer for nextfile() */
char *nullargv[] = { "-", NULL }; /* dummy argv for case of no args */
char *inname;                /* filename for messages etc. */
long lineno;                 /* line number for messages etc. */
FILE *in = NULL;             /* current input file */

extern void error(), exit();
#ifdef UTZOOERR
extern char *mkprogname();
#else
#define mkprogname(a)    (a)
#endif

char *nextfile();
void fail();

/*
 * - main - parse arguments and handle options
 */
main(argc, argv)
int argc;
char *argv[];
{
    int c;
    int errflg = 0;
    extern int optind;
    extern char *optarg;
    void process();

    progname = mkprogname(argv[0]);
```

```

while ((c = getopt(argc, argv, "xxxd")) != EOF)
    switch (c) {
        case 'xxx':      /* xxx meaning of option */
            xxx
            break;
        case 'd':        /* Debugging. */
            debug++;
            break;
        case '?':
        default:
            errflg++;
            break;
    }
if (errflg) {
    fprintf(stderr, "usage: %s ", progname);
    fprintf(stderr, "xxx [file] ...\n");
    exit(2);
}

if (optind >= argc)
    argvp = nullargv;
else
    argvp = &argv[optind];
inname = nextfile();
if (inname != NULL)
    process();
exit(0);
}

/*
- getline - get next line (internal version of fgets)
*/
char *
getline(ptr, size)
char *ptr;
int size;
{
    register char *namep;

    while (fgets(ptr, size, in) == NULL) {
        namep = nextfile();
        if (namep == NULL)
            return(NULL);
        inname = namep;      /* only after we know it's good */
    }
    lineno++;
    return(ptr);
}

/*
- nextfile - switch files
*/
char *
nextfile()
{
    register char *namep;
    struct stat statbuf;
    extern FILE *efopen();

    if (in != NULL)
        (void) fclose(in);

```

```

    namep = *argvp;
    if (namep == NULL) /* no more files */
        return(NULL);
    argvp++;

    if (STREQ(namep, "-")) {
        in = stdin;
        namep = "stdin";
    } else {
        in = fopen(namep, "r");
        if (fstat(fileno(in), &statbuf) < 0)
            error("can't fstat '%s'", namep);
        if ((statbuf.st_mode & S_IFMT) == S_IFDIR)
            error("'%' is directory!", namep);
    }

    lineno = 0;
    return(namep);
}

/*
 - fail - complain and die
 */
void
fail(s1, s2)
char *s1;
char *s2;
{
    fprintf(stderr, "%s: (file '%s', line %ld) ", progname, inname, lineno);
    fprintf(stderr, s1, s2);
    fprintf(stderr, "\n");
    exit(1);
}

/*
 - process - process input data
 */
void
process()
{
    char line[MAXSTR];

    while (getline(line, (int)sizeof(line)) != NULL) {
        xxx
    }
}

```

C program, separate input files

```

/*
 * name - purpose xxx
 *
 * $Log$
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

```



```

#define MAXSTR 500          /* For sizing strings -- DON'T use BUFSIZ! */
#define STREQ(a, b)        ((*a) == *(b) && strcmp((a), (b)) == 0)

#ifdef lint
static char RCSid[] = "$Header$";
#endif

int debug = 0;
char *programe;

char *inname;                /* filename for messages etc. */
long lineno;                /* line number for messages etc. */

extern void error(), exit();
#ifdef UTZOOERR
extern char *mkprograme();
#else
#define mkprograme(a)    (a)
#endif
void fail();

/*
- main - parse arguments and handle options
*/
main(argc, argv)
int argc;
char *argv[];
{
    int c;
    int errflg = 0;
    FILE *in;
    struct stat statbuf;
    extern int optind;
    extern char *optarg;
    extern FILE *efopen();
    void process();

    programe = mkprograme(argv[0]);

    while ((c = getopt(argc, argv, "xxxd")) != EOF)
        switch (c) {
            case 'xxx':    /* xxx meaning of option */
                break;
            case 'd':    /* Debugging. */
                debug++;
                break;
            case '?':
            default:
                errflg++;
                break;
        }
    if (errflg) {
        fprintf(stderr, "usage: %s ", programe);
        fprintf(stderr, "xxx [file] ...\n");
        exit(2);
    }

    if (optind >= argc)
        process(stdin, "stdin");
    else

```

```

        for (; optind < argc; optind++)
            if (STREQ(argv[optind], "-"))
                process(stdin, "-");
            else {
                in = fopen(argv[optind], "r");
                if (fstat(fileno(in), &statbuf) < 0)
                    error("can't fstat '%s'", argv[optind]);
                if ((statbuf.st_mode & S_IFMT) == S_IFDIR)
                    error("'%'s' is directory!", argv[optind]);
                process(in, argv[optind]);
                (void) fclose(in);
            }

        exit(0);
    }

/*
 * - process - process input file
 */
void
process(in, name)
FILE *in;
char *name;
{
    char line[MAXSTR];

    inname = name;
    lineno = 0;

    while (fgets(line, sizeof(line), in) != NULL) {
        lineno++;
        xxx
    }
}

/*
 * - fail - complain and die
 */
void
char *s1;
char *s2;
{
    fprintf(stderr, "%s: (file '%s', line %ld) ", progname, inname, lineno);
    fprintf(stderr, s1, s2);
    fprintf(stderr, "\n");
    exit(1);
}

```

Make file

```

# Things you might want to put in ENV and LENV:
# -Dvoid=int          compiler lacks void
# -DCHARBITS=0377     compiler lacks unsigned char
# -DSTATIC=extern     compiler dislikes "static foo();" as forward decl.
# -DREGISTER=         machines with few registers for register variables
# -DUTZOOERR          have utzoo-compatible error() function and friends
ENV = -DSTATIC=extern -DREGISTER= -DUTZOOERR
LENV = -Dvoid=int -DCHARBITS=0377 -DREGISTER= -DUTZOOERR

# Things you might want to put in TEST:

```

```

# -DDEBUG          debugging hooks
# -I.              header files in current directory
TEST = -DDEBUG

# Things you might want to put in PROF:
# -Dstatic='/* */'  make everything global so profiler can see it.
# -p              profiler
PROF =

CFLAGS = -O $(ENV) $(TEST) $(PROF)
LINTFLAGS = $(LENV) $(TEST) -ha
LDFLAGS = -i

OBJ = xxx
LSRC = xxx
DTR = README dMakefile tests tests.good xxx.c

xxx:    xxx.o
        $(CC) $(CFLAGS) $(LDFLAGS) xxx.o -o xxx

xxx.o:  xxx.h

lint:   $(LSRC)
        lint $(LINTFLAGS) $(LSRC) | tee lint

r:      xxx tests tests.good    # Regression test.
        xxx <tests >tests.new
        diff -h tests.new tests.good && rm tests.new

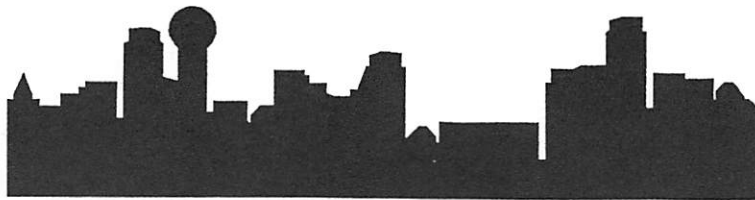
# Prepare good output for regression test -- name isn't "tests.good"
# because human judgement is needed to decide when output is good.
good:   xxx tests
        xxx <tests >tests.good

dtr:    r $(DTR)
        makedtr $(DTR) >dtr

dMakefile:  Makefile
        sed '/^L*ENV=/s/ *-DUTZOOERR//' Makefile >dMakefile

clean:
        rm -f *.o lint tests.new dMakefile dtr core mon.out xxx

```

The Integration Toolkit and the Unison Real Time Operating System

P. Kim Rowe, D. Graham, A. Donenfeld
Multiprocessor Toolsmiths Inc.
P.O. Box 13365 Kanata, Ontario
Canada K2K 1X5
(613) 838-3457

B. Pagurek
Dept. of Systems and Computer Engineering
Carleton University, Ottawa, Ontario
Canada K1S 5B6

ABSTRACT

The Integration Toolkit is a cross development environment for developing real-time embedded applications. It provides five main components. Composer is an intelligent system builder. CopyCat, is a host based simulator for the Unison real-time operating system. Remedy is a real-time system level debugger for Unison applications. A broad set of support tools for downloading, and file transformation completes the toolkit. This paper provides a discussion of the merits of real-time Unix approaches. It discusses the features of the Integration Toolkit and the Unison Real-Time Multiprocessor Operating System as a real-time Unix substitute.

Introduction

Real-time systems development projects are now faced with mounting problems. The result is late projects, poor software quality, cost overruns, and ineffective risk management. Now, real-time system developers are turning towards computer aided software engineering (CASE) to solve these problems.

Traditionally, real-time Unix extensions have taken two different approaches. One approach links a Unix host to a target running a real-time operating system, using a communications path between the systems. This is a good approach in many ways, but invariably, the programming environment in the target is completely foreign. Others have added real-time extensions to Unix; however, the fundamental primitives and models of Unix do not support real-time multitasking programming well. The communication, synchronization, tasking models, and device handling required for quality real-time system development is simply not available.

Real-time operating systems have additional problems. Very poor program development environments, lack of transparent multiprocessing capabilities, lack of debugging and testing support, and a requirement for target hardware before integration begins, have all created problems.

This paper discusses the Integration Toolkit [3,4,5,6,7,8,9,10,11] and the Unison Real-Time Multiprocessor Operating System [6,9], one of its main components. First, it considers the motivation behind

real-time Unix approaches, and how these tools perform. Secondly, the elimination of these problems by the Integration Toolkit and Unison is presented. Finally, the features, functions, and benefits of the Unison Real-Time Multiprocessor Operating System are presented.

Real-Time Unix Systems

Real-time Unix systems were originally conceived to overcome some of the inherent limitations of Unix for real-time applications. The idea was simply to exploit the open nature of Unix and add the necessary features to support real-time scheduling, and fast device handling.

The benefits of this approach closely emulate the benefits of Unix itself:

- a) Unix is an open system, ideally suited to extension.
- b) C language is ideal for real-time work.
- c) A broad set of high quality software development tools are available.
- d) Well defined and understandable models of computation are available.
- e) The combination of the models and the tools produces a high quality development environment.
- f) It is relatively vendor independent, and it is portable.
- g) Newer versions include networking, and window support.

For all the strengths of this approach, real-time enhancements to Unix have met with limited success. This is attributed to the inherent limitations of Unix for real-time work.

- a) a) Unix is large. Many real-time problems require small solutions.
- b) The enhancements are not carefully integrated into the conceptual models of Unix, eliminating a large part of the benefit.
- c) c) Real-time systems must handle a broad set of unusual devices as part of the application. Unix is not well suited to handling devices which do not conform to its I/O models, or users adding new devices simply and quickly.
- d) Source code licensing is expensive for Unix. This makes kernel enhancement too expensive. Furthermore, many users don't want to absorb the risk associated with being totally dependent upon the vendor.
- e) The message passing primitives, multiprocessing features, and device support required to make the system useful would require a substantial investment. It is not available as a supported product.
- f) Real-time I/O support is poor.
- g) Modern real-time systems require multiprocessing support.

The developers of real-time embedded systems recognized these limitations. They often opted for another solution to this problem; cross development and a real-time kernel. A broad set of host computers, and a broad set of real-time kernels are used today.

Several vendors have introduced products using this solution. Typically, they are host independent, and sold in ROM form. Unix host to real-time target communication links are part of the offerings.

The advantages of this approach are:

- a) The application development is host independent.
- b) The kernel may be small and well understood.
- c) Source licensing is inexpensive.
- d) Real-time device support is simple and efficient.
- e) Real-time device addition is simple.
- f) New feature addition is easy.

The disadvantages of this approach are:

- a) Vendors have not integrated development support. Users diagnose defects using an in circuit emulator, print statements, and/or assembly language.
- a) In house construction of the transformation, testing, debugging, and downloading tools results in inadequate tools.
- b) Transparent multiprocessing support is not available.
- c) Primitives are often poorly designed, and complex.
- d) The models of I/O and computation are completely unique to the system in question. This

creates a significant learning curve, and the need to work in two different environments.

- e) Prototyping and/or integration require hardware.
- f) Networking support is lacking.
- g) Specialized device support is lacking.
- h) User portability is deficient.

The Integration Toolkit and the Unison Real-Time Multiprocessor Operating System provide a solution which overcomes all of these problems. It provides a Unix model compatible real-time operating system, with complete cross development support, and an extensive library of device servers. The host environment is primarily Unix.

The Integration Toolkit

Multiprocessor Toolsmiths Inc. offers an innovative cross development environment called the Integration Toolkit. This toolkit simplifies and streamlines the design and integration phases of real-time system software development. Its use results in better quality products, produced in less time, with both cash flow and profit improvements.

The Integration Toolkit is a highly integrated set of tools for building, testing, and debugging real-time uniprocessor and multiprocessor systems. Its main goal is to provide programmers and real-time systems developers a high quality, cross development environment which reduces development and maintenance time, reduces resources, improves quality, and reduces development and maintenance risks.

The Integration Toolkit uses the latest innovations in software engineering. It relies upon the latest research into real-time operating systems, integration environments, and CASE methods. It is a sophisticated package which has been proven in developing real-time embedded systems for a broad set of applications. It can be simply integrated into current environments, and adapts well to changing industry trends. Industrial automation, military systems, automated testing, aerospace systems, simulation, medical systems, and animation systems are ideal application areas for the toolkit.

The Integration Toolkit offers an integrated, comprehensive set of tools to build real-time embedded systems. Sophisticated tools like the Remedy System Level Debugger [3,7,9], and the CopyCat simulator [8,9], are the second most effective means to improve your development. These tools improve system quality, reduce integration time, reduce system cost, and minimize integration risks.

The toolkit supports a networked development environment. It provides target system configuration management, high bandwidth downloading, and flexible information exchange. The time savings, cohesive project team building, and effective resource utilization this produces is of substantial importance.

The Integration Toolkit development tools run on a heterogeneous set of host computers. This set

includes IBM compatibles running MS-DOS or PC-DOS, Unix work stations, and Unix systems (figure 1). This broad set of host hardware and software provides the means to evolve your environment to adjust to a changing world. The ability to improve resource utilization, reduce purchase risk, and increase cash flow are also beneficial.

The Integration Toolkit supports a broad set of target hardware, including all industry standard buses, and processors. This flexibility to adapt to new target environments is a principle benefit. It allows users to rejuvenate products, or target new markets using existing applications software.

The Integration Toolkit consists of five main components. The Unison Operating System provides a real-time multiprocessor operating system facility. CopyCat provides complete Unison simulation. Remedy is a system level debugger for Unison, and Composer, an automatic system builder. The set of support tools provides downloading and other utilities [10]. This comprehensive, integrated, support environment provides the means for you to achieve substantial reductions in design time, integration time, and system cost. Using the toolkit also improves software quality, reduces development risks, and improves product adaptability.

The data flow between the four basic toolset components defines the tool interfaces (figure 2). The use and function of each tool is detailed below.

Composer, the configuration generator, automates the processes of porting Unison to new target hardware, producing makefiles to build applications, and building libraries for different target hardware configurations. The high degree of automation allows inexperienced users to use the full power of the environment without extensive training, saving time and resources.

Composer provides high level descriptions of target systems. It cuts out a substantial portion of the documentation required to provide multiple target systems to applications programmers, and provides the means for automatic makefile generation. The users benefit through time savings, and cost savings, as well as the quality improvement it produces.

CopyCat, the host based simulator for Unison applications, improves productivity by providing a high level environment to develop and debug concurrent programs in the host environment. It uses Unison semantics of message passing, and hardware simulation. This provides the means to start integration before the hardware is available, to eliminate target hardware, to eliminate downloading delays, and to control the environment more precisely. All of these features either save time, save resources, reduce risk, or improve quality.

CopyCat also provides support for structured testing and debugging. The approach suggested involves using CopyCat as an interim step when building a Unison application. All the logical errors

are eliminated on the host system under simulation. More detailed evaluation, particularly of hardware/software interaction, requires the target system. This approach saves time, and money, and produces a higher quality delivered product.

CopyCat provides complete source level debugging, message tracing, and task display in the host environment (figure 3). This set of features provides improved control of the application simulation, and the high level abstractions necessary to debug the application quickly and easily. The user accomplishes the debugging faster, at lower cost, and with improved quality results.

The Unison Operating System provides communication, synchronization, resource management and time management facilities. In addition, an extensive collection of device support software modules for building real-time embedded multiprocessors and uniprocessor systems are available. Software subsystem reuse is the most effective method to improve your development. It improves system quality, reduces your development time, reduces your system costs, maximizes your product flexibility, and minimizes your development risks. Unison features will be discussed in more detail in the following section

Remedy, the Unison application debugger, allows you to debug in real-time. With Unison running on the target, and Unix or MS-DOS running on the host, Remedy provides system level debugger features and high level abstractions of the target processing not offered by any other debugger. The high level abstractions and system level features substantially reduce debugging time, and debugging costs.

Remedy provides a dynamic display of tasks as they execute on the target, complete with the parent/child task creation hierarchy (figure 4). This dynamic view of the target provides a direct window into the application to understand the current state of the system. The time and resource savings that result are considerable.

Remedy host/target communications uses an ethernet, RS232 link or by shared memory. This provides the flexibility necessary to adapt to different development environments, and meet different project needs.

Remedy is an ideal monitoring system too. It tracks system operation, and displays its internal workings. This provides users with a tool which can reduce the maintenance costs of the system considerably, and teach people about the system faster.

Remedy completely replaces in circuit emulators for target system software development. This saves significant resources, particularly for multiprocessor applications, and results in much more profitable projects.

The Integration Toolkit uses mouse and menu based display systems, simple, comprehensive documentation, and on line help facilities. These features insure that the toolkit is easy to learn, and easy to

use. The net resource and time savings are substantial.

Training and structured methods are an integral part of the product. For the design of Unison systems in a language independent fashion, "Systems Design with Ada" [1] is available (figure 5). The structured testing methods suggested closely emulate "The Art of Software Testing [2]." They provide the means to optimize the use of the toolset, producing productivity improvements, quality improvements, and reduced risk.

The Integration Toolkit comes with complete support, including training, telephone support and emergency services, maintenance support, porting services, and field trouble shooting support. This comprehensive set of services insures that you may introduce the toolkit into your organization easily and effectively, and it will continue to meet your needs.

The Unison Operating System

The Unison Real-Time Operating System consists of a real-time kernel, and a broad set of device drivers called servers. A complete set of integrated development tools is available using the Integration Toolkit.

The key features of Unison include:

- a) Unix compatible I/O models
- b) Unix compatible networking
- c) transparent multiprocessing
- d) an open architecture
- e) multilingual capabilities, including direct support for Ada
- f) user portability
- g) an extensive device support library, which may be customized to suit new devices
- h) real-time performance
- i) dynamic binding, and dynamic resource management
- j) support for standards

Unison uses Unix compatible I/O models for most devices. This compatibility means that Unix programmers can easily program Unison applications with a minimum of training, saving time, saving resources, improving quality, and reducing risk.

Unison uses Unix compatible networking models from 4.3bsd. This compatibility again means that Unix programmers can easily program Unison applications with a minimum of training. Time savings, resource savings, quality improvements and risk reduction result.

The transparent multiprocessing offered by Unison makes programs involving many tasks independent of the number and type of underlying processors. This gives users the ability to easily customize systems, or upgrade systems to meet new processing demands. This flexibility is valuable during

development, during maintenance, or simply to meet a variety of customer needs. It saves time, saves resources, improves quality and reliability, and reduces risk.

The open architecture Unison offers supports the addition of new features. This supports customization to meet specialized needs. Risk reduction, time savings, and cost reductions result.

Unison has multilingual capabilities, including direct support for Ada. The current set of supported languages include C, Pascal, and Fortran in addition to Ada. The non-concurrent languages use similar primitives to add concurrency. This insures that Unison will provide an evolutionary path to an Ada environment and that existing software may be reused. This provides both the short term, and long term features necessary to evolve your environment.

Unison is user portable to new processor cards, and even new processors. The primary requirements for porting are a C compiler, and Unison source code. This gives the user additional flexibility to meet new customer demands by simply porting existing application software to new hardware. This flexibility reduces risk considerably, and saves both time and resources.

The extensive library of tried and proven device servers is an integral part of Unison. The set currently includes udp, tcp/ip, tty_server, file_server, block_server, calendar_clock, robot arm server, nfs_client, xwindows_client, and a variety of other servers. It provides a means to reuse software components in order to save time, save resources, improve quality, and reduce risk.

The Unison Operating System provides essential real-time embedded system features. Message timeouts and calendar clock features improve exception handling. Very high performance messaging, and transparent multiprocessing, minimizes system optimization, and saves both time and money.

Low interrupt latency times and priority based, preemptive, natural break scheduling insures demanding real-time deadlines are achieved. As a result, the user has more room for error during system design. It also eliminates the expensive optimization and redesign which must occur when deadlines are missed using a less responsive system. The savings in both time and resources are clear.

Unlike Ada, Unison provides dynamic binding of task identifiers and task entries. Like Ada, it provides dynamic allocation of all resources. This dynamic binding makes Unison ideal for handling situations where the environment is uncertain, and the application needs to assign resources in varying situations. The user benefits from this by being able to reduce the total system resources, and through the development and maintenance of a smaller application.

Support for standards is an integral part of the Integration Toolkit and the Unison Operating System. The motivation for Unix compatibility is

standardization to eliminate unnecessary learning. The direct Ada support is the result of standardization on new real-time languages. Support for tcp/ip, nfs, xwindows, and other defacto standards will produce similar benefits. The net result is a system which uses an integrated set of standards to solve application problems. The users benefit from this approach through increased portability, reduced learning, and considerably reduced risk.

In summary, Unison provides a complete environment and support for developing real-time applications. It supports Unix compatible application development, in a variety of languages, with a high degree of flexibility, adaptability, and portability. It is ideal for meeting demanding real-time constraints. This approach saves time, improves quality, reduces costs, and reduces risk.

Summary

The review of the existing real-time Unix approaches illustrated a number of strengths and weaknesses. The feature set provided by the Integration Toolkit eliminates these weaknesses without sacrificing any of the strengths. It provides substantial benefits to users.

The Integration Toolkit relies upon the latest research in real-time operating systems, CASE tools, and real-time software engineering methods. Standards are an integral part of the toolkit. This approach insures that users remain in the mainstream of real-time development, yet they can exploit the latest technology. This new technology, produces more profitable projects, faster, at lower risk, and with improved quality.

The Unison Operating System provides a core set of features to support real-time multiprocessing. It emphasizes real-time response, software reuse, and Ada semantics of message passing. This approach reduces development time, saves money, improves quality and reduces risk.

References

- [1] Systems Design with Ada, R.J.A. Buhr, Prentice Hall 1984.
- [2] The Art of Software Testing, Glenford Myers, , 1979
- [3] Remedy, A System Level Debugger, P.K. Rowe, B. Pagurek, IEEE RTSS 87, Dec. 3-5, 1987, San Jose, CA, IEEE 1987.
- [4] A Real-Time Multiprocessor Development Environment, P.K. Rowe et al, CASE 87 Advance Papers, Index Technology Corp. 1987.
- [5] The Integration Toolkit Overview, Multiprocessor Toolsmiths 1987.
- [6] The Integration Toolkit Unison User's Guide, Multiprocessor Toolsmiths 1987.
- [7] The Integration Toolkit Remedy User's Guide, Multiprocessor Toolsmiths 1987.
- [8] The Integration Toolkit CopyCat User's Guide, Multiprocessor Toolsmiths 1987.

The Integration Toolkit Development Environment

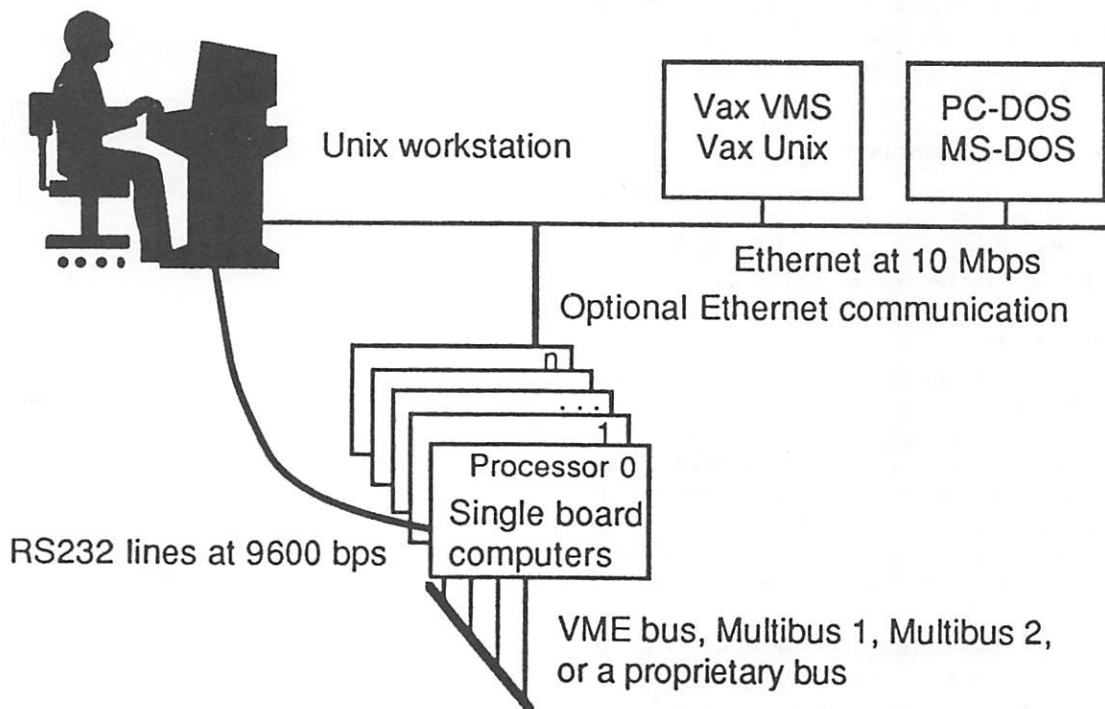


Figure 1: *The Integration Toolkit development environment consists of PC compatibles, VAX computers, Unix work stations, and a broad set of target hardware, all networked by ethernet or more primitive means. It provides an evolutionary approach to introduce new technology into your environment, minimizing disruption, learning, risk, and cost.*

Components of the Integration Toolkit

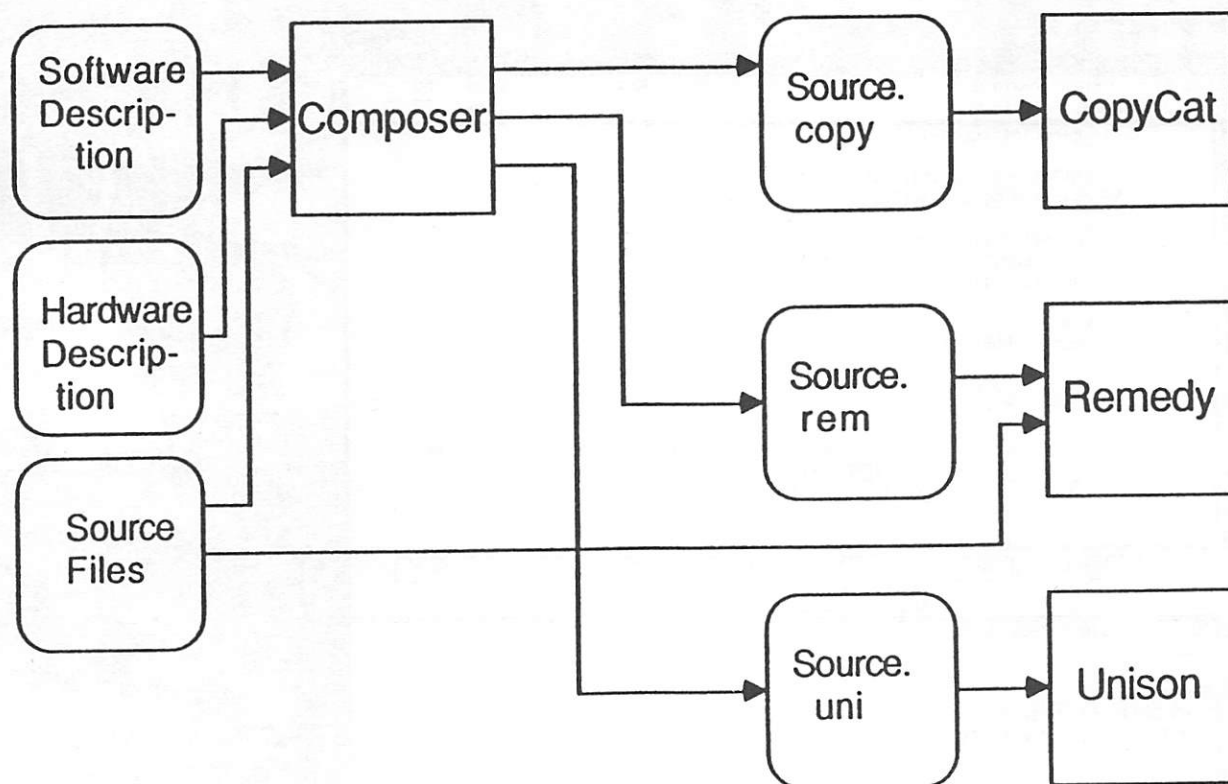


Figure 2: *The Integration Toolkit has four principle components, Composer, CopyCat, Remedy, and Unison. Composer generates systems from high level descriptions of the hardware and software. CopyCat is a host simulator for the Unison real-time embedded multiprocessor operating system. Remedy is a graphical, system level debugger for the host and target development environment.*

CopyCat with dbxtool

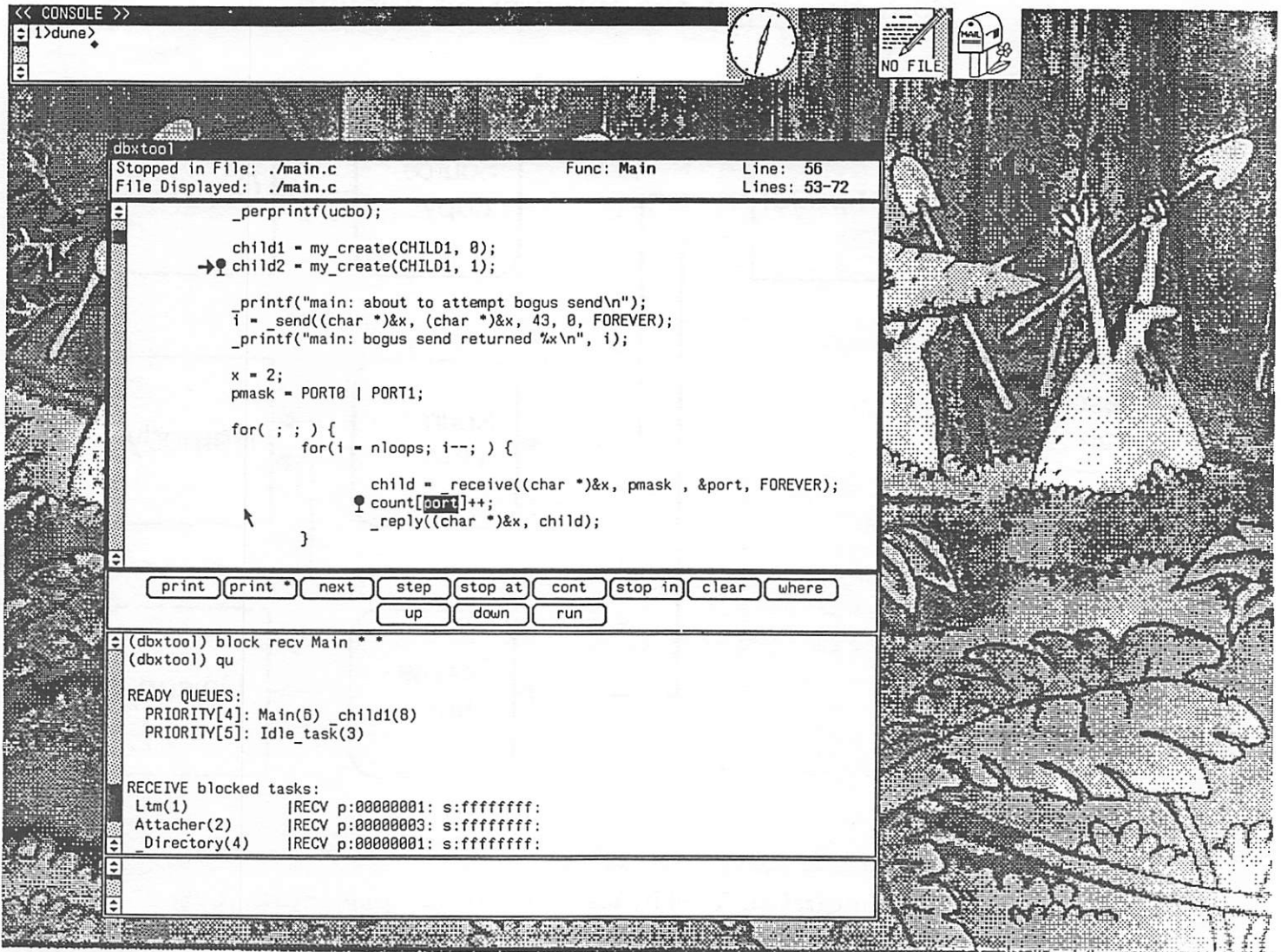


Figure 3: *CopyCat*, the *Unison* Operating System simulator, provides complete uniprocessor target emulation including interrupt generation and handling. *CopyCat*, in conjunction with *dbxtool* or other symbolic debuggers, provides a facility to completely integrate software components quickly and easily, without the need for target hardware.

The Remedy Debugger Host Interface

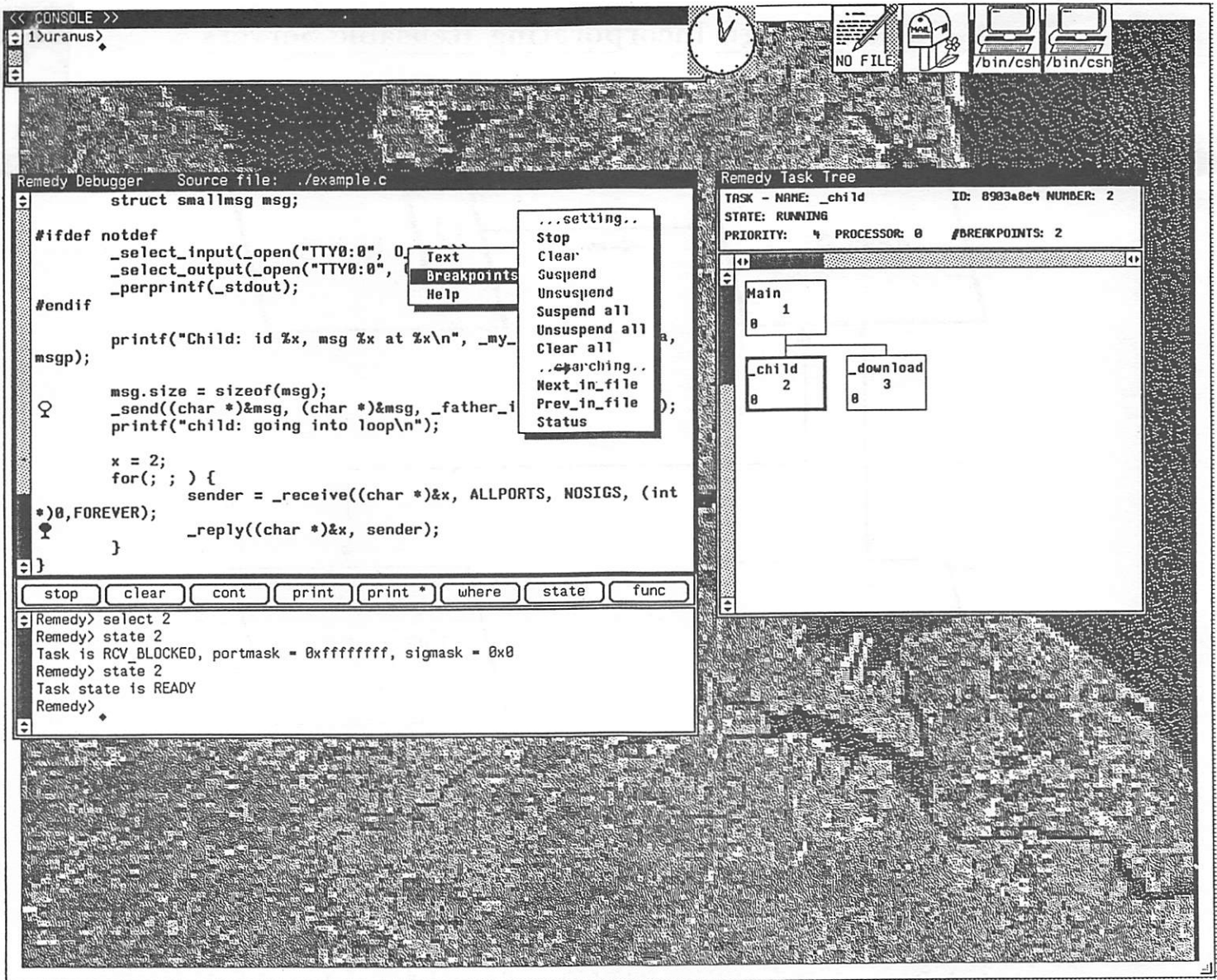


Figure 4: The Remedy debugger displays the dynamics of task creation and destruction in real-time in the task frame on the right. Breakpoints for the currently selected task are shown in the source window, with the suspended breakpoint shown as hollow. When the menu is pulled up, only the valid options for breakpointing are allowed.

A Unison Design Incorporating Reusable Servers

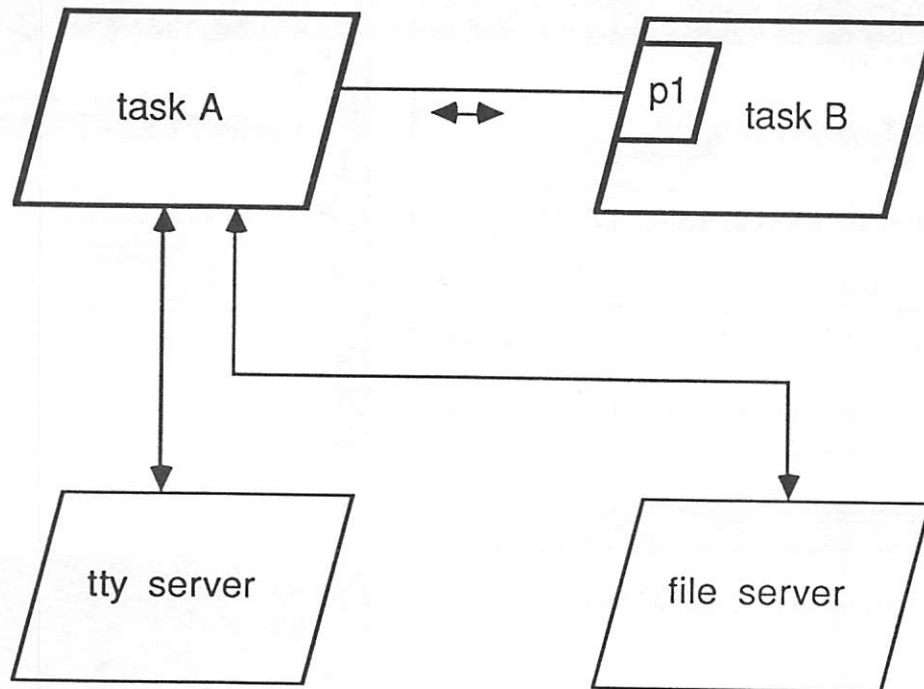
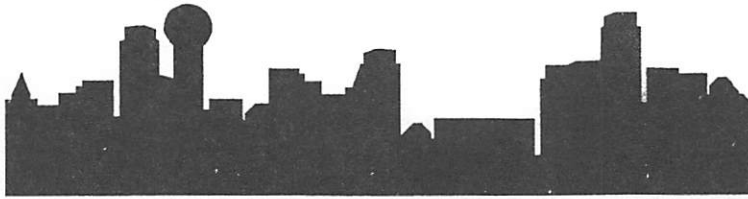


Figure 5: The solid edged parallelograms are tasks, and the servers, which are being reused, are represented by the lighter edged parallelograms. The arcs represent communication between the elements. A send from task A to task B on port 1 is shown. File I/O is shown between A, and both the file server and the tty server.



USENIX Winter Conference
February 9-12, 1988
Dallas, Texas

Process Migration in UNIX Networks

K. I. Mandelberg
V. S. Sunderam
Emory University
Atlanta, GA 30322
gatech!emory!km
gatech!emory!vss

ABSTRACT

In a distributed computing environment, the ability to migrate running processes is considered desirable for load balancing and other reasons. This paper describes a scheme to realize process migration on a network of Unix workstations. The mechanism used addresses and resolves some of the limitations of similar projects, and demonstrates that a user-level implementation is capable of providing a sophisticated and efficient migration facility. The migration software has been successfully implemented and some preliminary results and analyses are presented.

Introduction

Process migration refers to the capability in distributed computing environments to dynamically relocate processes between processors. The primary motivation for providing this facility is load balancing; when the total workload is evenly distributed, other benefits such as greater throughput and reduced overheads are obtained. There are also other reasons that justify the need for a migration facility. It may be necessary to halt a processor or logically detach it from the network - process migration would allow most of the processes to continue execution on a different processor. Certain special facilities may only be available on some processors; a process could migrate to such a processor to access those facilities when it needs to. An urgent process could be migrated to a lightly loaded processor in order to complete in a shorter period. A multitasking application might migrate some of its subprocesses to different processors and thereby achieve greater parallelism.

Process migration normally requires considerable cooperation and support from the underlying operating system software. Owing to this fact, successful migration facilities are encountered mostly in truly *distributed* environments such as Charlotte [1] or the V-system [2]. However, independent processors (particularly workstations) connected by high speed local area networks are much more common than full-blown distributed systems. In these environments, each processor is controlled by an independent operating system and network services as well as other forms of communication are located, for the most part, outside of the executive software.

The objectives of this work were to investigate the feasibility of supporting process migration on a network of independent processors and, further, to

attempt this without any special support from or modifications to the operating system kernel. The goals were to design as complete a facility as possible that could be implemented and used in an efficient, simple, and straightforward manner. In addition, we have attempted to overcome the restrictions and limitations on migration that diminish the usefulness of similar systems. The main effect of this goal has been to enable existing user programs to benefit from migration, without the need for any modifications or even recompilation.

Process migration in almost all implementations is performed by taking a "snapshot" of the process on a processor and reinstating the snapshot on a different processor. In the Unix context, this implies obtaining an image of the process' data and stack areas, *execing* the process on a different processor and restoring the original data and stack area states. In the case that the resurrection is done on the same processor, these steps are tantamount to a checkpoint and restart ability. For long running programs, this facility in itself can be beneficial and mitigates the cost of machine failures. Some of the basic assumptions and restrictions in our design of a migration scheme are presented in the next section following which the critical design and implementation issues are described. Preliminary results and experiences are then reported with our analysis of the situations under which the facility is likely to be most effective and useful. A section on future directions and improvements that are desirable concludes this paper.

Goals, Issues, and Restrictions

The process migration facility described here has been targeted at computing environments consisting of object-compatible workstations connected by a high-speed local network. The software to implement migration has been developed under Unix 4.2BSD + NFS and our work is particularly targeted at networks of Sun workstations. Typical workstation applications often include a significant number of programs that are long running and expend substantial resources; migrating these dynamically can lead to even distribution of workload as well as increased turnaround. Typical examples are text-formatting programs, simulation exercises, and large numerical computations.

One of the major goals of this project was to build the migration facility entirely as user-level processes as opposed to incorporating the system either partially or wholly in the Unix kernel. This contributes to greatly reduced development, testing and maintenance efforts; more importantly, installation and use is very straightforward. One of the drawbacks of such a scheme is that the operation of the facility is likely to be slower, but this tradeoff was considered acceptable particularly keeping in mind the targeted applications. One kernel change is however necessary to allow the migration of processes that use a system supplied value - namely the process id - but this is only encountered in rare circumstances and is an optional part of the system.

Another important goal was to avoid the use of specialized stubs i.e. additional interfaces to system and library calls that are left running on the originating machine when a process migrates. Other systems [3] have used this approach which necessitates recompilation of application programs and can lead to substantial overheads. Furthermore, "object-only" software is excluded from migration if such a scheme is adopted. An important facet of our design is that

predetermination of migratable processes is not necessary. It is possible, at any point in the execution of a process, to suspend it and reinstate it in such a way that it then joins the pool of candidates for migration.

The mechanism by which processes are migrated in our system is distributed between cooperating software on the originating and destination processors. However, decisions concerning which processes to migrate and when to migrate them are made by a single agent that is external to the migration mechanism. This achieves a clear separation of function and allows changes in migration policy and algorithms to be incorporated easily and independently of already executing processes.

In the current implementation, a process, to be migratable should not (a) perform I/O on non-NFS files (b) spawn subprocesses or (c) utilize pipes and sockets. We believe the first requirement is not restrictive - considering the wide acceptance of NFS as a de-facto standard and the targeted operating environments.

Terminal Interface

One of the most important issues to be considered while relocating a process is the method of severing its interaction with the user terminal and reconstructing this interface after migration. This should be done without loss of any data in transit between the process and the terminal as well as with minimal interference to the user. Described below is the scheme adopted to handle terminal interfaces in our implementation.

Unix processes are 'attached' to a terminal known as the 'controlling tty' which normally serves as the standard input and output device for the process. The controlling tty may either be a hard device or, as in common in window systems, a pseudo-tty (pty) which is a software simulation of a tty between two processes. In either case, the kernel maintains

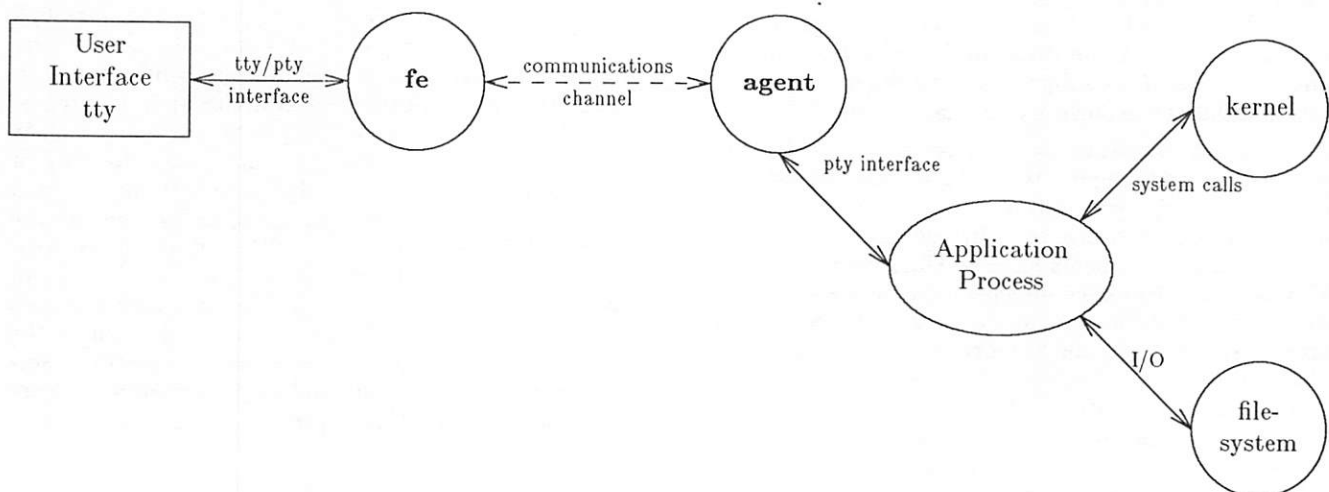


Figure 1: Modified Unix process interfaces

input and output queues for the tty which contain data that may be partially processed or consumed, depending on the modes of the tty. Processes interact with the tty using the same filesystem interface as disk files.

However, if a process is detached from its tty by simply severing the tty interface, it will terminate abnormally unless it has foreseen this condition. Further, the data in the tty queues will be lost. Therefore, in order to migrate a process, an alternative terminal interface must be provided that insulates the process as well as the user from these effects. Such an interface has been developed to support migration; it also has the added feature of using a uniform mechanism irrespective of the processor on which the tty is a device and of the processor on which the process is executing.

The interface consists of two processes that communicate using a stream connection. The **fe** process is a front-end that transfers data between the terminal and the stream connection. The **agent** process is connected to the other end of the stream and transfers data between the stream and a pseudo-tty that is set up between itself and the application. The **agent** also listens on a control port for migration related requests. A diagrammatic view is shown in figure 1. It is evident that such a scheme parallels that used in remote logins; in fact the **fe-agent** mechanism is an extended, more efficient version of the remote login model.

While it appears that such a scheme may incur excessive overheads, in practice no effects are noticeable. Terminal emulators in window systems [4] always use similar schemes without loss of performance, even though the applications there are terminal I/O intensive, which our targeted applications are not. Moreover, our system permits processes to be initiated directly from a tty and later, when required, to be migrated and reattached via the **fe-agent** interface, thereby alleviating some of the overheads.

Process Initiation and Monitoring

In order to be able to select processes for migration and to determine target processors, information concerning process and processor resource utilization needs to be maintained. Processor load information is already maintained by other software¹ and is directly used. Resource utilization figures for each process that is a candidate for migration is maintained in a globally available log file and consists of the virtual and physical sizes of the process and its CPU usage². At regular intervals, the controlling **agent** process obtains these figures for its protege and updates the log file.

The log file contains one entry for each candidate process. Each entry, in addition to resource utilization

¹e.g., `rwho`, `rwhod`. If more detailed processor data is required, this software can be easily adapted to provide it.

²The policy algorithm itself is discussed in 5.4.

figures, also specifies:

- a unique identifier for the process and an initiation timestamp. Logfile access is exclusive; at initiation the **agent** assigns an unused identifier and records this in the log.
- the state of the process, indicating whether the process is "connected" (to its controlling tty), "disconnected", or "migrating".
- the hostname of the processor on which it is currently executing, and the control port of the corresponding **agent** process.
- the complete pathname for the executable file for the process.

A process that has been predetermined to be a candidate for migration should be invoked by the user through the **fe-agent** mechanism. The only user action that is required is to prepend "fe hostname" to the normal command line. If `<hostname>` is omitted, the local host is assumed. The **fe** process initiates an instance of **agent** on the target host, establishes a stream connection to it, and forwards the remainder of the command line to it, as well as local environment information. The remote **agent** process sets up the identical environment and executes the application process after creating the pty interface. It also initializes the log file entry for the process.

Migration Mechanism

When a process is to be migrated, the controlling **agent** receives a migration request specifying the hostname of the target machine. The controlling **agent** halts the process then initiates the migration, which is performed in as described below.

Obtaining the Process Image

As mentioned, the basic mechanism employed is that of obtaining a snapshot of the process on the originating machine and reconstructing the process on the target machine using this snapshot. Apart from the terminal interface, Unix processes also interact with the kernel by means of system calls and with the filesystem. Most Unix system calls are either atomic or idempotent or both. Therefore, no special handling is necessary in this regard for most system calls. Certain system calls however, are stateful, and the process' description as well as kernel table information must be reconstructed after the process has been migrated. These aspects as well as the preservation of I/O states for a migrated process are dealt with in later sections. In the Unix context, a process image consists of:

- (1) text, data, and stack segments and auxiliary kernel data structures.
- (2) open files for the process along with current pointers and modes.
- (3) the state of its controlling terminal and any data in the terminal queues.

To obtain a snapshot of the process, Unix already provides the core dump mechanism. However,

this mechanism can only be invoked by the kernel itself and normally results in process termination. Programs that obtain core dumps externally (e.g. *gcore*) are not guaranteed to work always and therefore an alternative mechanism was required. In our system, a series of *ptrace* system calls is used to read the data and stack address spaces for the process and a core file is created. Kernel data structures that contain additional information regarding the process (i.e. the *proc* structure and the *ublock*) are obtained by reading kernel virtual memory and are also saved.

Process files

In addition to the process image, state information pertaining to files currently open by the user-process must also be obtained. A Unix process normally performs I/O on two types of files - disk files and the terminal to which it is attached.

To be able to restore the context of disk files used by a process, several file attributes must be extracted and reproduced when the process is reinstated on a different processor. Under Unix, one of these attributes (*viz.* the pathname) is difficult to obtain owing to Unix filesystem semantics. A pathname loses significance once a file has been opened - the process requesting the open is returned a file descriptor using which all I/O is carried out. Other software requiring the reconstruction of a file's pathname do so by making kernel/system call modifications or by exhaustive search - but none of these methods is guaranteed to work. For our purposes, the pathname itself was not important; what was required was a way to be able to preserve a pointer to the file so that the migrated process could continue to access it.

The solution to this problem was to create a link to the file and use the pathname of the link after migration as an access point to the file. Thus when the process snapshot is being created, a link (with a special name) is created to each file that is in use by the process. This involves traversing kernel data structures (by reading kernel virtual memory) to determine the NFS "filehandle" for the file and executing a *link* NFS call to create the link. Auxiliary information concerning the file such as the mode and current position of the file pointer are extracted and saved during this step.

Unix and NFS semantics also restrict the creation of a link to a file on the same filesystem. Therefore, a special directory is used on each filesystem in which the links are created; filesystem mount information is used to determine the appropriate filesystem for each file processed.

Terminal devices

Most Unix processes use at least one terminal device for I/O; this is normally the "controlling tty" for the process. As explained in the previous section, the scheme used in our project allows processes to be detached from their controlling tty and later re-

attached. Since this mechanism, by nature, will work even when the tty and the process are on different processors, the migration mechanism does not need to incorporate special provisions in this regard.

However, at the time of obtaining the process snapshot, the tty may be in a state where I/O is partially complete. Unix tty devices have three internal queues where varying amounts of data may be present depending upon the tty settings, type-ahead, and program controlled reads, writes & flushes. Emptying these queues (by external means) prior to migration is possible only in some cases. Therefore, the migration system extracts the contents of the queues - once again by reading various kernel data structures - and preserves this along with other process state information. The tty modes and settings are also obtained and preserved.

Signal Handling

Unix provides system calls using which processes may "block", "catch" or "ignore" signals. Information pertaining to signals (e.g. which signals are being caught & what their handler routines are) is maintained by the kernel in the *proc* structure and the *ublock*. For the most part, restoration of signal information simply involves duplicating the appropriate values in these two structures in the corresponding structures of the migrated process.

Reinstating the Process

The process is reinstated on the target machine by first creating a new process from the original object code file. However, this new process is allowed to execute only after the snapshot is superimposed on it, thereby achieving the effect of "continuing" the original execution. The controlling **agent** first performs a *fork* system call to create a new process that is a copy of itself. Prior to *execing* the user process object file, disk files that were open in the original process are opened and their pointers and modes correctly set. Before the user process is loaded, a *ptrace*(*TRACEME*) system call is done; this causes the user process to pause prior to execution. The controlling **agent** restores the state of the process to what it was prior to migration and then allows the execution to proceed.

The restoration itself is straightforward and essentially involves writing the user process' stack and data space from the saved core image. This is again done by *ptrace* calls as is the resetting of processor registers to their old values, particularly the program counter. Other state information such as signal masks are reconstructed by writing the *proc* and *ublock* kernel data structures. A few aspects of this reconstruction, however, require special handling and these are discussed below.

Restoring Tty Queues

The partially processed data in the original tty queues is required to be put back in the queues of the tty for the new process. Since the kernel allocates queue buffers only when actual I/O is done, and since the queues are empty when the new process is first created, it is not possible to reconstruct the queues directly. In our scheme however, the process communicates with its tty via the controlling **agent** and in fact has a pseudo-terminal (pty) interface with the **agent**. The **agent** process is therefore capable of performing I/O on both the master and slave sides of the pty. To reconstruct the pty queues, the **agent** forces the creation of kernel buffers by a series of write and read operations on both sides of the pty with a particular combination of pty modes. Once the buffers have been allocated, the original queue data is written into these by writing to kernel virtual memory.

Process Data Area

When a new object file is loaded with the *exec* system call, it is allocated a data area of a compiler estimated size. Most large programs, however, expand their data areas during execution by using the *sbrk* call. Therefore, after a process has been migrated and reinstated on the target machine, its data area has to be expanded to what it was when the migration was initiated. This is not possible to do externally; in other words, only the process itself can expand its own data area. In order to do this, the controlling **agent** places a section of object code (that contains an appropriate *sbrk* call) on the user processes' *stack* and forces the user process to execute this code. Once this is done, the data from the snapshot is written with a series of *ptrace* calls.

Signals & Interrupted System Calls

At the time of obtaining the snapshot, the process may have been executing a system call since some calls are not atomic. In particular, this can happen with the *read*, *write* (on slow devices), and *sigpause* calls. If the snapshot had been taken under such conditions, correct process continuation will be possible only if the system call is performed again. The controlling **agent** therefore checks for these conditions and adjusts the program counter as needed to reissue the system call.

After the migrated process has been reinstated, the controlling **agent** detaches itself from the process and allows it to continue execution.

Preliminary Results

The process migration facility described has been successfully implemented and has been in use at the authors' installation. Several "real" applications have been subjected to migration and the results are encouraging.

Migration Policy

At this stage, the policy used to determine candidates for migration and their target processors is somewhat primitive. However, this decision and the initiation of migration is external to the mechanism that implements it - therefore, modification of policy algorithms and their testing can be carried out independently. At present, the controlling **agent** maintains and periodically updates resource utilization data for its protegee process. This data consists of (a) the current virtual size of the process (the sum of its stack, data & text sizes), (b) its current resident set size, and (c) the percentage of CPU time it has utilized while resident. These measurements, although instantaneous, provide a reasonable estimate of the memory and processor requirements for the process.

The external migration agency or migration daemon (**migd**) is a user process that executes its algorithm cyclically after remaining idle for a parameterized time period. The algorithm consists of first determining if there is a marked difference in load between each pair of processors participating in the scheme. This is done using the three (time decayed) load average values already maintained by the "remote who" daemon. A source host and target host are selected if there are hosts such that:

$$\max(targ_{15}, targ_5, targ_1) + 1 < source_1$$

and

$$source_1 > 1$$

where $targ_n$ is the n-minute load average for the target processor.

This ensures that processes from hosts which are not CPU saturated (load average < 1) are never migrated and also attempts to obtain a "worst case" prediction of the target host load average after migration. At present, memory availability and utilization for processors is not considered.

If viable source and target hosts are found, the global log file is analyzed for processes presently executing on the source processor. That process with the highest CPU use and ratio of virtual to real memory is selected for migration; provided that the process is, in fact, currently receiving fewer CPU cycles than it requires. Once such a process has been identified, the **migd** process conducts a migration by first initiating a dump request on the controlling **agent** followed by a reinstate request on the target machine.

Exercises

The interface and migration mechanisms as well the migration policy have been tested on three real application programs. They are a unix shell, a compute intensive combinatorics program, and a symbol manipulation package and were selected as representing various classes of long running processes. The

tests were performed to check correct migration by exhaustive trials and to measure the costs involved in migration.

The shell program is relatively small and does not, by itself, consume CPU cycles. It is mostly I/O bound and contributes little to the load on a processor. As was expected, it was never a candidate for migration. When induced to migrate by artificial means, it was found to require an average of 13 seconds for migration - 8 seconds to acquire a snapshot and 5 to reinstate the process. Processes such as shells are therefore not good candidates for migration; however, since migration preserves shell variables and the environment, it may be useful in certain circumstances.

The combinatorics program is heavily compute intensive and determines certain properties of graphs by exhaustive search. Typical running times for this program range between 20-25 days on lightly loaded Sun 3's. Owing to the high CPU utilization of this program, it was always a strong candidate for migration. It was found that this process migrated far too frequently, except when potential target hosts were themselves executing other compute intensive processes. This, of course, was a failing of the migration policy; which has since been adapted to foresee such unstable conditions. The cost of migration in this case averaged 23 seconds - 14 to obtain the snapshot and 9 for reinstating the process.

This program is a typical example of very long running scientific and numerical applications. Such programs perform relatively little I/O, almost never to an interactive terminal. When the hosts participating in migration are substantially and unevenly loaded, it is evident that migrating such processes benefits both the process and the throughput of all processors on the whole. However, the extent to which this is true depends a great deal on the migration policy and the cost of migration; extensive tests are under way to obtain more specific, quantitative results. The ability to migrate such long running processes presents another, very valuable advantage. Particularly in a workstation based development environment, it is common for hosts to reboot frequently. When the total running time of a program is of the order of tens of days, migration allows such processes to continue uninterrupted execution. At present, processes are manually migrated off hosts that will reboot; however, it is straightforward to automate this process.

Migration exercises were also performed on a very large algebraic manipulation package. This program is compute intensive when performing symbolic calculations and has an extremely large (5.5MB) data space. Migrating such programs is very beneficial when, on hosts with limited real memory, swapping and paging overheads become exorbitant. However, the time for migration in this case was found to be 650 seconds (400 for the snapshot, 250 for reinstating the process); thereby diminishing the value of

migration. Again, such processes are worth migrating only under certain circumstances e.g. when there is considerable disparity between hosts in terms of available and utilized memory, or when it is necessary to bring down the processor on which the process is executing.

During the course of these experiments, it was noted that the time required to migrate processes was a linear function of the size of their core image. An average of 14 seconds elapses for each 100K of core image size including both phases of the migration. This limitation is primarily a filesystem bottleneck, particularly severe under NFS; the processing overheads for migration are minimal. We are investigating an alternative scheme where the process snapshot is directly transferred to the target host without creating a disk copy. Preliminary tests show that the elapsed time for each 100K of core image is reduced to about 3 seconds.

Conclusions and Future Work

This project has shown that it is possible to implement process migration to advantage on a network of independent workstations. The most noteworthy aspects of our work are that the implementation is completely free of kernel modifications and that existing programs need not be changed or even recompiled in order to be migratable. As yet, concrete measurements and precise quantification of the benefits are not available; however, the potential value of the system seems considerable. Further work is required in a few areas to improve the applicability and effectiveness of the system and these are listed below.

Cost of migration

The time that elapses during migration is excessive, thereby mitigating the advantages of migrating processes that either run for small durations or those that have large memory requirements. However, it is interesting to note that the CPU time required for migration is less than 5 percent of this elapsed time. It has already been established that avoiding the creation of the snapshot on disk will reduce this very markedly and such a scheme is being designed.

Migration Policy

The migration policy is at present extremely primitive and is based on readily available but instantaneous and incomplete data on both the characteristics of processes and the state of processors. The policy algorithm requires reworking to ensure that migrations are based on more accurate data and that unstable situations are not created. However, an overly sophisticated algorithm could itself be expensive and inappropriate.

Elimination of Restrictions

The present implementation does not migrate processes that have subprocesses or those that use communications sockets. Although most CPU intensive applications (towards whom migration is primarily targeted) do not fall into this category, handling such processes would greatly enhance the applicability of the system. Compilers and shell pipelines would benefit by this inclusion which is planned in the next stage of the project.

References

- [1] Y. Artsy, H-Y. Chang, R. Finkel, "Processes Migrate in Charlotte", *University of Wisconsin - Computer Sciences Department*, Technical Report #655, August 1986.
- [2] D. Cheriton, "The V kernel: A software base for distributed systems", *IEEE Software* 1,2, 1984.
- [3] M. Litzkow, "Remote UNIX: Turning Idle Workstations into Cycle Servers", *Proceedings of the Summer 1987 Usenix Conference*, Usenix Association, June 1987.
- [4] R. Scheifler, J. Gettys, "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2, April 1986.



A Process Migration Implementation for a Unix System

Rafael Alonso
609-452-3869
Kriton Kyrimis
609-683-1691
Princeton University
Department of Computer Science
Engineering Quadrangle
Princeton, NJ 08540
allegra!princeton!alonso, alonso@princeton.edu
allegra!princeton!kyrimis, kyrimis@princeton.edu

ABSTRACT

In this paper we describe the implementation of a process migration mechanism under version 3.3 of the Sun UNIX operating system. Processes that do not communicate with other processes and that do not take actions that depend on knowledge of the execution environment (such as the process id), can be moved from one machine to another while running, in a transparent way. This is achieved by signaling a process to stop, saving all the kernel and memory information that is necessary to restart the process, and then, by using this information, restarting the process on the new machine. This new functionality requires minor kernel modifications as well as the creation of a new signal and a new system call.

Introduction¹

Process migration is the capability to move a process that is running on a certain machine to another, without interrupting its execution. This is a useful tool to have, as it can be used in various ways, ranging from system applications such as load balancing and process checkpointing, to applications for individual users such as moving a process from a machine that is about to go down, to another.

The interface to this mechanism should be transparent to the process that is being migrated. Specifically, the process should not know anything about the process migration mechanism and, after it is moved to another host, it should continue its execution as if it were still running on the original machine. The performance of this mechanism may vary depending on the purpose for which it is used. If it is used to even the load among a number of machines (load balancing), then it must introduce little overhead to the system, and must be able to move processes between machines quickly, requiring time comparable to that of the time required to load and start a program. On the other hand, if process migration is used for moving individual CPU intensive tasks in order to achieve better performance for those tasks, or to remove important tasks from a machine that is

about to be halted, then it is acceptable for the mechanism to have a lower performance.

In this paper we start by mentioning previous implementations of process migration and describing our implementation environment, explaining how it is different from that of these other implementations. Then, in Section 4, we present the user interface to the process migration mechanism and provide examples for it. First we present the interface as a casual user might see it and then as a programmer who wants to write new applications that use this mechanism. In Section 5 we describe the additions and modifications that were made to the Sun 3.3 UNIX kernel to add the process migration capability to it. In Section 6 we present our measurements of the performance of the new kernel and the applications that were written on top of it and in Section 7 we discuss the limitations of our system. In the eighth section we give a brief consideration of implementing process migration outside the kernel and, in the last section, we present our conclusions.

Other implementations

There are only four other implementations of process migration of which we are currently aware:

In the DEMOS/MP operating system¹ all interaction with the kernel is achieved by exchanging messages with it. This extends to the kernel itself, which can use the message mechanism to communicate with

¹This work has been supported by New Jersey Governor's Commission on Science and Technology Contract 85-990660-6, and grants from IBM and SRI corporations.

the kernel of another machine and send the state of a running process to that other kernel. The original process is then replaced with a degenerate one, which simply forwards all messages for the original process to the new one.

The Locus distributed UNIX system² attempts to distribute all the resources that a program is using, including the CPU, among all the machines in a network, in order to achieve network transparency of all resources. This system provides the *migrate* system call to change the execution site of a program that is already running.

The V-System³ also provides a network transparent environment. In this system process migration is implemented with the *migrateprog* command which copies the state of a process from one machine to another.

Finally, in the Sprite operating system,^{4,5} process migration is implemented by moving a process on another machine, but having those system calls that have different effects if executed on different machines (like get time of day, get process id), executed on the original machine, by exchanging messages between the process and the kernel of the original machine. In this way, although a process may be physically located on a different machine, it is actually working under the kernel with which it started its execution.

All of these implementations have relied on special features of the operating system that create a distributed environment and make it relatively easy for two machines to cooperate in moving a process from one to the other. However, "ordinary" operating systems such as ours, a UNIX implementation which evolved to its current state but was not designed for distributed use, do not have such features and implementing process migration under them is more difficult.

Since UNIX does not provide means of communication between two kernels, our implementation was somewhat limited in scope, in the sense that not all processes can be migrated. Apart from badly behaved processes (which are discussed in Section 7), the main limitation is that our mechanism does not provide for the migration of sockets. In our discussion of our implementation's limitations we argue that this does not render it useless.

Implementation environment

Our implementation was made on Sun 2 workstations running version 3.3 of Sun O.S., which is a derivative of the Berkeley 4.2 BSD version of the UNIX operating system. The machines were connected to each other and a file server by a 10 Mbit Ethernet, which provided the physical medium for moving processes from one machine to another. Each workstation's local files, along with the files that reside on the file server, are available on every machine by means of Sun's Network Filesystem^{6,7} (NFS). On our particular system we followed the convention of the 8th research edition of the UNIX

operating system of mounting the root directory of a machine to the "n" subdirectory of the root directory of all other machines.

User level description

Our process migration system provides certain new commands that the user can use to move processes from one machine to another. These commands are implemented by using a new signal and a new system call that our kernel provides. Knowledgeable users can use these new features directly to write their own process migration commands.

User commands

Most of the implementation code for process migration is at the user level. By this we mean that all commands that have to do with process migration are user applications. However, if the available commands do not fit a specific need, users can easily write their own substitutes (see Section 4.3).

We have provided the following three commands, which should cover most of the common cases:

- *Dumpproc* - terminate a process (kill it) dumping to disk all the information that is necessary to restart it. The process is determined by specifying its process id with the *-p* option. For security reasons, only the superuser or the owner of the process can kill a process in this way.
- *Restart* - restart a process that was killed on some host with the *dumpproc* command. The process is specified by its process id using the *-p* option, and the name of the host on which the process was dumped is specified with the *-h* option (the default is the current machine). The process will be restarted on the host on which the command was given and at the terminal (or window) on which the command was typed. All files that had been open when the process was dumped will be available to the restarted process with the correct access modes and offset. Terminal modes are preserved, so that visual applications such as screen editors can be restarted properly. Again, for security reasons, only the superuser or the owner of the original process can restart a process.
- *Migrate* - move a process from one machine to another. This is simply a combination of the two previous commands and can be used to avoid having to go to another terminal to type the *dumpproc* or *restart* command. The process id of the process to be moved is specified again with the *-p* option, the name of the host from which the process is to be moved is specified with the *-f* (from) option, and the name of the destination host can be specified with the *-t* (to) option (the default for both hosts is the current machine). The process is restarted on the terminal (or window) that the command was typed, even though the host that the process will run may not be the one on which that terminal is connected. *Migrate* calls *dumpproc* and *restart* internally, by using the remote execution command *on*, if

necessary.

Writing new applications

As we mentioned in the previous section, it is possible to write new commands that handle process migration in a different way from that of the three commands that we described. To do so, the following two items are available to the programmer.

A new signal, SIGDUMP

When a process receives this signal, the process is terminated, and all the information that is necessary to restart it will be dumped to disk. This information is in the form of three files, which are placed in the `/usr/tmp` directory, named `a.outXXXXX`, `filesXXXXX` and `stackXXXXX`, where `XXXXX` is the process id of the dumped process.

The first file is an executable obtained by dumping the text and data segments of the process, and prepending a suitable header that will make UNIX recognize the file as an executable. This file can be executed as an ordinary program. The result of such an execution will be similar to running the original program from the beginning, except that all static variables will be initialized to the values that they had when the process was killed. This gives us, incidentally, the `undump` utility for free. (This utility creates an executable file by combining an executable and a standard core dump.)

The second file contains all the information that is not needed by the kernel to restart the process, but must be used at user level if the process is to restart successfully. This information consists of:

- a "magic number" for identification purposes (arbitrarily set to octal 445).
- the name of the host on which the process was currently running at the time it was killed.
- the absolute path name of the current working directory.
- for each entry in the open file table of the process, an indicator specifying whether the entry refers to an open socket, open file, or is unused. For open files, this indicator is followed by the absolute path name of the file, the file access flags, and the file offset. Since the process migration mechanism does not currently support sockets, no extra information is kept in the case of a socket.
- the terminal flags.

All path names in this file have been constructed by combining the names given by the process to the kernel whenever it changed directory or opened or created a file, and resolving any references to the current or parent directories. This means that symbolic links are not resolved and this may cause problems when trying to reopen a file when restarting the process. Consider for example a file on a machine called *classic*, named `/usr/foo`. If `/usr` is a symbolic link to `/n/brador/usr` (i.e., `/usr` is mounted via NFS to the `/usr` directory of the machine named *brador*),

then `/usr/foo` is actually `/n/brador/usr/foo`. Now, let us assume that a program opens this file and is then terminated with the `SIGDUMP` signal. When the program is restarted, this file must somehow be reopened. One way would be to prepend `/n/classic` to the old name and open `/n/classic/usr/foo`. Because of the symbolic link, however, this would actually be `/n/classic/n/brador/usr/foo`. Unfortunately, NFS does not allow this syntax, so using this file name would not produce the desired result.

The way to solve this problem is to resolve symbolic links before files are reopened. This can be done by iteratively using the `readlink()` system call to resolve all symbolic links in a pathname (this is the main reason for using the `dumpproc` program instead of the `kill` command of the shell - `dumpproc` modifies the `filesXXXXX` file so that for all file names contained in it, symbolic links are resolved and the string `/n/<hostname>` is prepended to the names of all local files).

The third file contains all the information that is required by the kernel to restart a process. This information consists of:

- A "magic number" for identification purposes (arbitrarily set to octal 444).
- The user credentials.
- The size of the stack when the process was terminated.
- The contents of the stack.
- The contents of all the registers.
- All the information kept in the user and process structures that is

A new system call, `rest_proc()`

This call is used to restart a process that was terminated using the `SIGDUMP` signal. It takes two arguments, the names of the `a.outXXXXX` and `stackXXXXX` files mentioned above. Its effect is to overlay the current process with a copy of the process from which the two files are created, resuming execution from the point where the process was killed. Normally, there is no return from this system call, as the process that invokes it is destroyed. If the system call does return, this means that either the system didn't have enough resources to create the new process, or that something was wrong with the two files (they did not exist or they had an incorrect format). Note that this system call is in many ways similar to the `execve()` system call.

Before issuing this system call to restart a process, a program should do the following:

- Set its real and effective user id to that of the old process.
- Change its current working directory to that of the old process.
- Open all files that were open when the old process was running, assigning the same file numbers that they had in that program. These files must be

opened with the correct access modes and positioned at the correct offset.

Implementation

To make the UNIX kernel capable of supporting process migration we had to make certain modifications and additions to it. In this section we start by describing the modifications we had to make in order to make the kernel keep track of certain information that we required, and then we describe the additional features we provide, which use these modifications to implement the process migration mechanism.

Kernel Modifications

The main problem in the Sun UNIX implementation is that the kernel does not keep enough information about a process' current working directory and open files to enable us to deduce in a non-trivial way what these files are. To overcome this, the kernel structures were augmented to include the names of these files in the following way:

One of the most important structures in the kernel is the *user* structure, which contains all the swappable information about the process that is currently being executed. A character string of fixed size was added to this structure, which contains the full path name of the current directory. After each successful call to the *chdir()* system call, we do the following: if the argument to *chdir()* is an absolute path name, it is simply copied to the user structure; if it is a relative path name, it is combined with the value of the old current working directory in the user structure and the result is copied back. This field is initialized when the first call to *chdir()* is made with an absolute path name, with the updating procedure being skipped if the field has not been yet initialized. Since such a call is made early on during the UNIX startup procedure at boot time, and new processes inherit this field from their parent, we conclude that this field is correctly maintained for all processes.

Information about open files is contained in an array of pointers to *file* structures, one for each of the maximum number of open files that is allowed. Each file structure has been augmented with a pointer to a dynamically allocated character string containing the absolute path name of the file to which it refers. The field containing the path name of the open file is initialized after either a successful *open()* or *creat()* system call. In either case, the file name is copied from the arguments of each of these system calls into the entry of the file structure that is associated with the file that is being opened. If the file name is a relative path name, its name is combined with the name of the current working directory in the user structure to create the required absolute path name.

The required memory for the string that contains the file name is obtained by calling the Sun 3.3 kernel's memory allocator. When the file is closed, using the *close()* system call, this memory is released. To make

sure that the pointer to the name always has a correct value (either null or a valid pointer), the kernel subroutine *falloc()*, which allocates new file structures, has been changed to initialize this pointer to a null value.

Kernel additions

Since the kernel now keeps track of all the information that we want to have about a process in order to migrate it, implementing the *SIGDUMP* signal is simply a matter of dumping the appropriate data from the kernel structures onto disk. The code is similar to that for the *SIGQUIT* signal, which causes a process to terminate, producing a core dump.

In standard UNIX, new processes are created by overlaying an existing process with the image of a new program, by using the *execve()* system call. This system call cannot be used as it presently exists to restart a process that was dumped with the *SIGDUMP* signal. This is because *execve()* initializes the stack and clears the registers. To overcome this, we have added the *rest_proc()* system call to our kernel, which has been built upon *execve()*. For this purpose, the *execve()* system call has been slightly modified, to check a global flag which, if set, indicates that it is called from within *rest_proc()*. In that case, instead of calculating how much initial stack to allocate for the process, based on the command line arguments and the environment, it simply allocates as many bytes as are indicated in another global variable, thus making it possible to allocate as much stack as the process that is being restarted had when it was stopped.

Using this, the *rest_proc()* system call works as follows:

- It opens the *stackXXXXX* file, checking access permissions and verifying its format by checking the magic number at the beginning.
- Reads the user credentials and the size of the stack from that file.
- Sets the global flag indicating process migration and sets the variable that indicates the desired stack size.
- Calls *execve()* to execute the *a.outXXXXX* file, with the environment set to null. (As the environment of the old process was stored in its stack, it will be automatically restored when the stack is read in.)
- Resets the variable indicating process migration, so that further calls to *execve()* will work properly.
- Sets the user credentials to those already read. The old credentials were used to execute the *a.outXXXXX* file, so that only the owner of the process or the superuser is able to do it.
- Reads in the contents of the stack and registers.
- Reads in the information on the disposition of signals, and establishes it as that of the current process.
- Returns. At this point, the process running is a copy of the old process.

Performance evaluation

In this section we present the various measurements we made on our implementation. First, we compare the performance of the system calls we have modified to that of the unmodified ones in the original UNIX kernel. Next, we compare the performance of the new *SIGDUMP* signal and the *dumpproc* program to that of the *SIGQUIT* system of the original UNIX kernel, which is similar in function. We then compare the performance of the new system call *rest_proc()* and the *restart* program to that of the *execve()* system call, which is the closest to the new system call that the original kernel had. Finally, we compare the performance of the *migrate* application in various instances, as compared to running the *dumpproc* and *restart* applications separately.

System overhead

As we mentioned in the implementation section, the *open()*, *creat()*, *close()* and *chdir()* system calls were modified to keep track of the names of the current working directory and all open files, inside the kernel. For the *open()/close()* system calls, we gauged the overhead by measuring the system CPU execution time of a program that opens and closes a certain file for a hundred times, both under the standard UNIX kernel and under our new kernel. Since the *creat()* system call simply calls the same internal routine that *open()* calls, with slightly different arguments, we did not consider it necessary to measure its performance. For the *chdir()* system call, we measured the overhead by measuring the system CPU time of a program that executed one hundred sets of three calls to *chdir()*, one with an absolute path name as an argument, one with the parent directory “.” as an argument and one with a path relative to the current directory “..”, in order to make certain that all cases of combining the new value of the current directory with the old one, kept in the user structure, were considered. The results are summarized in Figure 1, with the performance of the original UNIX kernel normalized to 1 shown on the left vertical axis and the actual times (average for one of *open()/close()* pairs or set of

three *chdir()* system calls) shown on the right vertical axis. Our measurements show an overhead of about forty per cent (44% for *open()/close()*, 36% for *chdir()*).

Dumping a process

Since the *SIGDUMP* signal that is used to stop a process in order to move it to another machine is so similar to the *SIGQUIT* signal, it is appropriate to compare the performance of the former to that of the latter. A program was started and then killed repeatedly in the following ways:

- By killing it with a *SIGQUIT* signal.
- By killing it with the new *SIGDUMP* signal.
- By killing it with the *dumpproc* application program.

Since we were measuring the performance of the process migration mechanism, and not that of the file system, the program was chosen to be a small one, but which could still be used to verify that our mechanism was working correctly. The program increments and prints three counters (a register, a static variable allocated on the data segment and a variable allocated on the stack). On each iteration it inputs a line and appends it to an output file. This program, was always killed after its first prompt for input.

For each case, we measured the CPU and real time required to kill the process. The results are summarized in Figure 2, where the performance of the *SIGQUIT* signal is normalized to 1. *SIGDUMP* requires roughly three times as much time (both CPU and real) as *SIGQUIT*. Considering that *SIGDUMP* produces three dump files instead of the one that *SIGQUIT* produces, the result is very satisfactory.

Dumpproc requires roughly four times as much CPU time and six times as much real time as the *SIGQUIT* signal. The extra CPU time is to be expected, as, in addition to using *SIGDUMP* to kill the process, the program has to modify the *filesXXXXX* file. The large discrepancy between CPU and real time can be explained by noting that the three files that are produced by *SIGDUMP* are created by the process that is

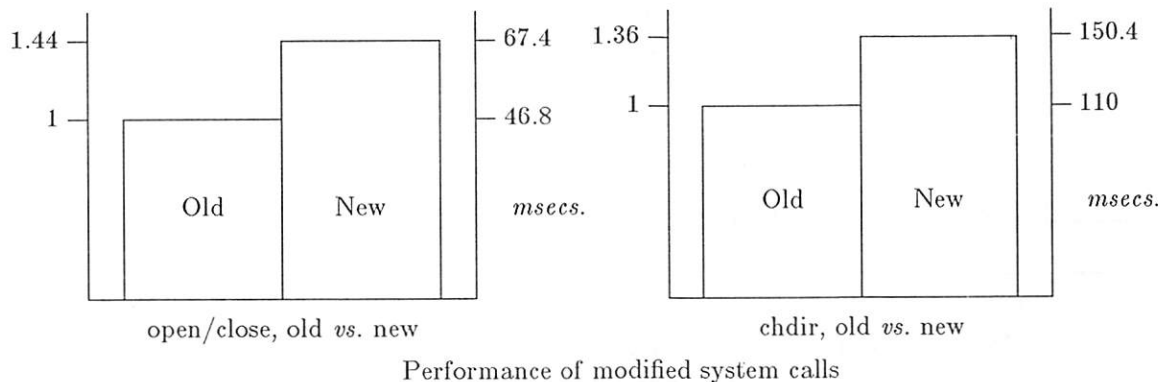


Figure 1

being dumped. When *dumpproc* tries to open the *a.outXXXXX* file, it has to wait until the kernel switches its context to that of the process being dumped, so that the file can be created, and then wait again until the kernel switches its context back to it.

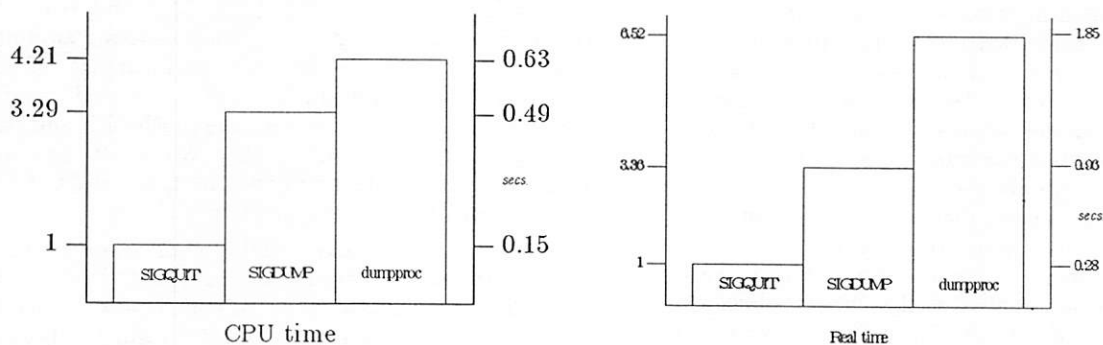
To avoid busy loops, *dumpproc* simply sleeps for one second after each unsuccessful attempt to open *a.outXXXXX* (aborting after ten tries). Since the order of magnitude of the times involved is that of executing a UNIX signal (about 0.6 seconds for killing our particular test program with *SIGDUMP*), we feel that the performance of the new signal is quite adequate.

Restarting a process

Since the *rest_proc()* system call that restarts processes that had been stopped on another host with the *SIGDUMP* signal is so similar to the *execve()* system call that executes new processes, it is again appropriate to compare the performance of the two, along with the *restart* application. For each case we measured the CPU and real time required by the system call or program in question to restart a test program (for *execve()* we measured the time required to execute the *a.outXXXXX* file). The performance of the system calls was obtained by adding timing code inside the kernel, as these system calls destroy the

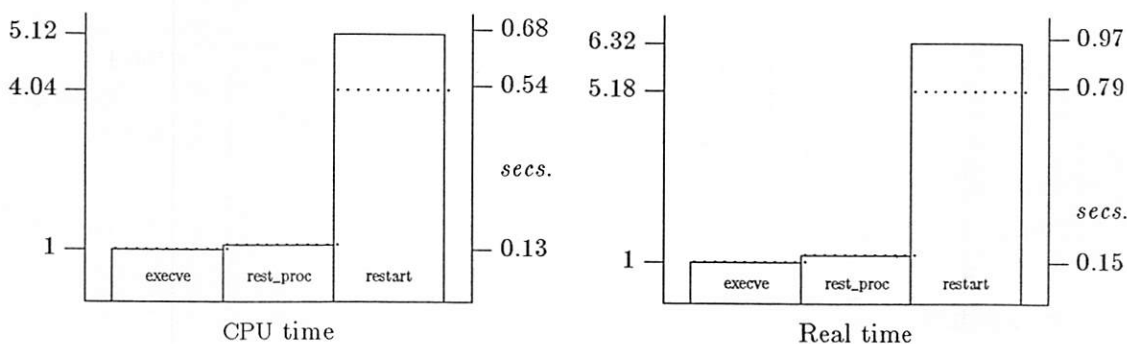
process that invoked them, making it hard to measure their performance at user level. The performance of *restart* was measured by timing its execution up to the point where it called *rest_proc()*, and adding to it the value already obtained by timing that system call. This is summarised in Figure 3, where the performance of *execve()* has been normalized to 1. The dotted line in *restart*'s bar denotes the relative contributions of *restart* itself and *rest_proc()*, which were measured separately.

We note that *rest_proc()* takes only slightly longer than *execve()*, which is entirely satisfactory. The *restart* application takes significantly longer (roughly five times more CPU time and six times more real time) than *execve()*. This can be justified by the fact that the application has to check the existence and verify the format of the three dump files and, most importantly, set that part of the process environment that can be set at user level, including the open files, which requires a large number of *open()* system calls. Nevertheless, considering the amount of work that is being done, the delay is not unacceptable, especially when we consider that our unit of measurement is the time required to execute a process, which, for our test program was less than 0.2 seconds, both in real and CPU time.



Relative performance of the *SIGQUIT* and *SIGDUMP* signals and the *dumpproc* application

Figure 2



Relative performance of the *execve()* and *rest_proc()* system calls and the *restart* application

Figure 3

Migrating a process

In addition to the *dumpproc* and *restart* applications, we have the *migrate* application which combines the actions of *dumpproc* and *restart*, so that the user need not move to another terminal to kill or restart his or her process.

Migrate has been implemented by executing the other two applications internally, by means of the remote execution command *on*, if any of those programs needs to be executed on a remote machine. *On* was used instead of *rsh*, the BSD remote shell command, because *rsh* may require as much as three times more time than *on* to establish a connection between two machines, and this would make the *migrate* command prohibitively expensive. The measurement results are summarised in Figure 4, where the performance of the *dumpproc/restart* combination is normalized to 1.

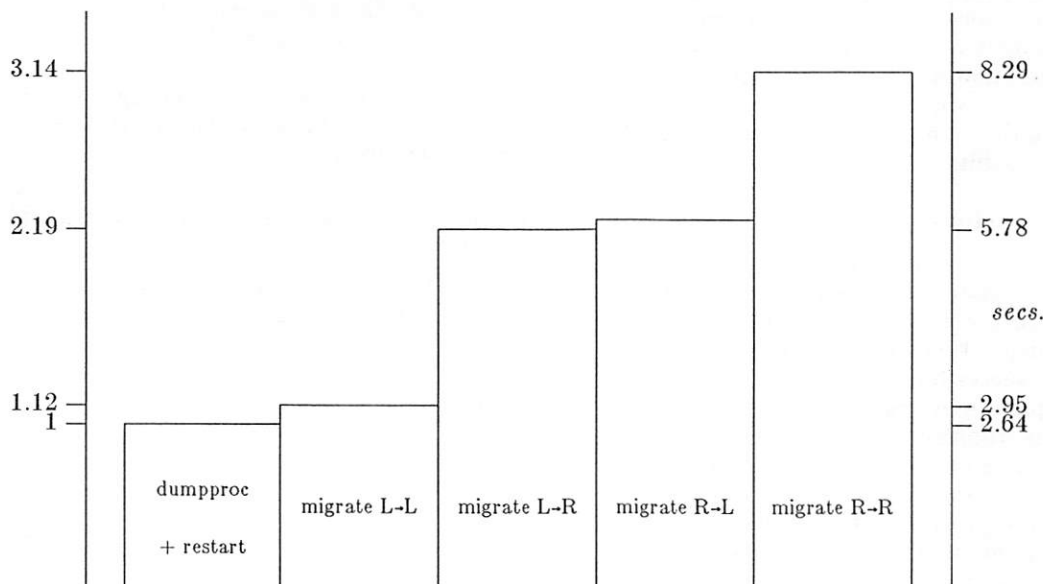
Limitations

Although our system has been fully implemented, not all processes can be migrated successfully under our current design. The main limitation is the inability to redirect pipes and sockets to the migrated process. The best we can do in our current implementation is to redirect socket I/O to a file, which is probably of little use. However, processes that qualify for process migration are those that have lots of CPU activity and little I/O activity and are probably running by themselves without communicating with other programs. This means that, even without preserving sockets, the process migration mechanism is still useful.

The other limitation has to do with programs that "know" things about their environment, such as

their process id or the name of the host in which they are running, and use this knowledge in ways that make their dependent on running on the same host (e.g. using their process id as part of a temporary file name, or making assumptions about the existence of special-purpose hardware, like a floating point processor, based on the hostname). One solution that could be implemented to solve the first of these two problems is to add an extra field for an old process id and maybe even an old host name in the user structure, and change the *getpid()* and *gethostname()*, system calls to return those new fields if the process has been migrated. This would require providing new system calls, that would return the real values, regardless of whether the process has been migrated or not. Programs that would use the new system calls would know of the existence of the process migration mechanism and would therefore be able to avoid transferring the problem of knowing things about their environment from one set of system calls to another. Programs that use the old system calls will be made to believe that they are running in their old environment. This would eliminate the temporary file problem, but will aggravate the problem of deciding what to do depending on the environment. Although processes that were migrated before making such decisions can run under the current system, they will make the wrong decision and crash if these modifications are implemented. However, there should be very few, if any, applications that fall in this category and such a modification is worth considering.

A further caveat is that processes that wait for one or more of their children to complete should not be migrated while waiting. When such a process is



Real time performance of the *migrate* application, compared to running the *dumpproc* and *restart* applications separately (L=Local machine, R=Remote machine)

Figure 4

moved to another machine, it ceases being the parent of what used to be its children, and waiting for them will produce undefined results.

A final point which must be mentioned even though it is not actually a limitation, is the fact that our system does not tolerate much heterogeneity. Processes can be migrated to a similar CPU or to one whose instruction set is a superset of that of the original machine. Doing otherwise would result in trying to execute machine instructions on the destination machine which the machine does not have.

Consideration of an implementation outside the kernel

Although our implementation requires making some modifications to the UNIX kernel, it is possible to implement a large part of it outside the kernel, and have it run as a user application. Specifically, the information that is dumped to disk with the new SIGDUMP signal can be retrieved by examining `/dev/kmem`, `/dev/mem` and the raw disks. This would take an inordinate amount of time, however, especially the part of determining the names of all open files. For this, we would have to scan the inode table of the device on which a file resides, trying to locate a directory entry that contains the inode number for that file in order to find the file's name, and then backtrack through the inode table up to the root of the file system, in order to determine the entire path name of the file.

As for the second half of the implementation, that of loading all this data back into memory, there does not seem to be any simple way of doing it, so we would still require some kernel support for this part. This leads to the conclusion that process migration requires kernel modifications in order to work. Since the minimal modifications required would produce a very slow process migration mechanism, we feel that our implementation, which requires kernel modifications for both the dumping and the restarting of a process, is the best solution.

Conclusions

We feel that our implementation of process migration has been successful. Programs that do not communicate with other processes and that do not take actions that depend on knowledge of their environment can be successfully migrated to other machines, with complete user transparency. This is achieved efficiently, as stopping a process and restarting it on another machine requires a time comparable to that of killing the process to obtain a core dump and then restarting the process at the beginning on another machine using the standard UNIX system calls.

Since our current implementation does not migrate processes that use sockets, the next step in our research will be to examine whether support for sockets can be added to our system. Another interesting subject, which is the focus of our current work, is

to use the process migration capability to implement and evaluate various load balancing algorithms, which, for the most part have never actually been implemented because process migration is not a common feature of operating systems.

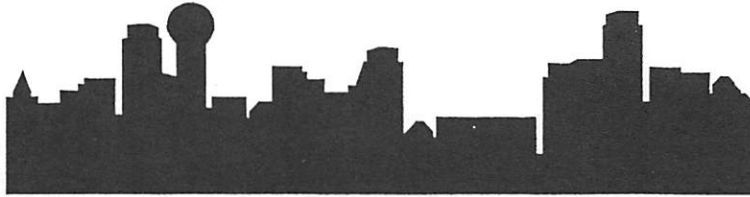
In retrospect, implementing our system was relatively easy, despite the fact that this was our first experience with the UNIX kernel. Most of our time was spent in understanding the workings of UNIX. Once such an understanding was achieved, coding our modifications and additions was relatively straightforward, due in part to the modularity of the Sun UNIX kernel and in part to the fact that most of our code was a relatively simple variation of existing kernel code.

Acknowledgements

We would like to thank Marc Staveley for assisting us in understanding the workings of the Sun UNIX kernel, and for his help in debugging our kernel modifications. We would also like to thank Larry Rogers for helping us in our first steps through the sources of the Sun operating system.

References

1. Michael L. Powell and Barton P. Miller, "Process Migration in DEMOS/MP," *Proceedings 9th ACM Symposium on Operating Systems Principles, Operating Systems Review* 17(5) pp. 110-119 (October 1983).
2. David A. Butterfield and Gerald J. Popek, "Network Tasking in the Locus Distributed Unix System," pp. 62-71 in *Proceedings of the Summer 1984 Usenix Conference*, ().
3. Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proceedings 10th ACM Symposium on Operating Systems Principles, Operating Systems Review* 19(5) pp. 2-12 (December 1985).
4. Fred Douglass, "Process Migration in the Sprite Operating System," U. C. Berkeley Technical Report (1987).
5. John Ousterhout, Andrew Cherenon, Fred Douglass, Michael Nelson, and Brent Welch, "An Overview of the Sprite Project," *login: The USENIX Association Newsletter* 12(1)(January/February 1987).
6. Mordecai B. Rosen and Michael J. Wilde, "NFS Portability," pp. 299-305 in *Proceedings of the Summer 1986 Usenix Conference*, ().
7. S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," pp. 238-247 in *Proceedings of the Summer 1986 Usenix Conference*, ().



Chad Hunter
The MITRE Corporation
Burlington Rd., Bedford Ma. 01730
harvard!linus!chad

Process Cloning: A system for duplicating UNIX processes

ABSTRACT

This paper presents process cloning, a method of duplicating a running, UNIX process, in order to migrate it to another machine in a network. This technique has been employed in a system for remote execution of processes on idle workstations. Such processes run only when no interactive users are logged in and the system load average is sufficiently small as defined by a workstation's owner. Otherwise, a machine is considered "in use" and remotely executed jobs are automatically suspended. Process cloning, however, allows a suspended job to migrate to another machine on the net, should one be available.

The process cloning system was implemented without modification to the operating system kernel, a strategy which has resulted in a simple, portable, and extensible system. Relatively few programming restrictions are imposed on the user; he need only refrain from accessing special files directly, using nonstandard file buffers, catching signals, locking files, and calling on the file control (fcntl) facility. Although the implementation is currently limited to childless processes with no inter-process communication, possible extensions will be discussed, as will the procedure for the capture and restoration of a process' I/O state, context, and executable image.

The same techniques involved in process cloning could also be used to perform check-pointing, a means of insuring the progress of a process by taking regular "snapshots" of it. Should anything, such as a system crash, interfere with the process, it could be recreated from the most recent snapshot. This type of fault tolerance also lends itself to a distributed processing environment.

Introduction

This paper describes "process cloning" a strategy for duplicating a running process. Cloning can best be understood from an historical perspective.

Our network at MITRE consists of workstations assigned to individual users, who, of course, do not continuously use their machines. The distributed batching system was developed to harness the consequent vast resource of wasted CPU time. It does so by enabling users to submit jobs, called "satellites", to any machine in the network for background execution so long as the machine is not in use. Machines become in use when users log on or the load average exceeds a predetermined level.

This first implementation of the batching system was able to run an integer factoring algorithm, the Multiple Polynomial Quadratic Sieve [1], distributed over a network of twenty-five Sun 3 workstations for more than one thousand CPU hours so effectively that it outperformed a similar algorithm run on a Cray-XMP/48. Still, we realized that an even more impressive performance could be eked out of the batching system, since executing satellites are put to sleep whenever a user logs on. Instead of sleeping, the

satellite had only to be moved to any idle machine on which it could then resume execution. Process cloning of course provided this liberation, and was developed as an extension to the batching system.

The implementation of process cloning was written without modifying the operating system kernel, which offers several advantages:

- Portability: Porting the system requires few parametric changes and rewriting only a single module of about 40 assembly instructions.
- Durability: Since the UNIX kernel is not changed, no proprietary source code is required; consequently, operating system upgrades should not interfere with cloning.
- Simplicity: Both the cloning process and the writing of clonable programs is simple: the first because a process transparently participates in its own cloning; the latter because only a small set of restrictions must be observed to ensure successful cloning.
- Extensibility: Since modification does not require knowledge of UNIX internals, and the system has been designed simply, it is easily expanded.

Process cloning is practical, for it provides

significantly greater utilization of the unused cycle resource in a typical network and imposes relatively few programming restrictions on the user. The success of this project not only demonstrates the viability of an implementation free of kernel modification, but also its advantages. Process cloning may find additional use in providing conventional process check-pointing as well as fault tolerance in a distributed processing environment.

The next two sections describe the distributed batching system, as it was before process cloning and after the extension of process cloning had been added, respectively. Subsequent sections discuss the use of process cloning and its underlying mechanisms, including the capture and restoration of a process' I/O state, context, and executable image. Possible extensions will also be examined.

The Distributed Batching System

The batching system is controlled by low overhead programs called "batch daemons," one of which resides on each machine in the network. "Host" programs, which must be linked with an "rcmd library," submit satellite programs to remote daemons for execution. This library serves as an interface between a host program and daemon and enables the former not only to submit satellites but also to communicate with them. Satellites, which must be linked with an "rcmdio" library if they are to be migrated, are any programs executable by the C-Shell, such as system commands or user programs.

In addition to overseeing the local execution of satellites, batch daemons maintain status information about themselves and their peers. One of the daemons, an omniscient master, helps the others garner this information. Periodically, it ascertains and then

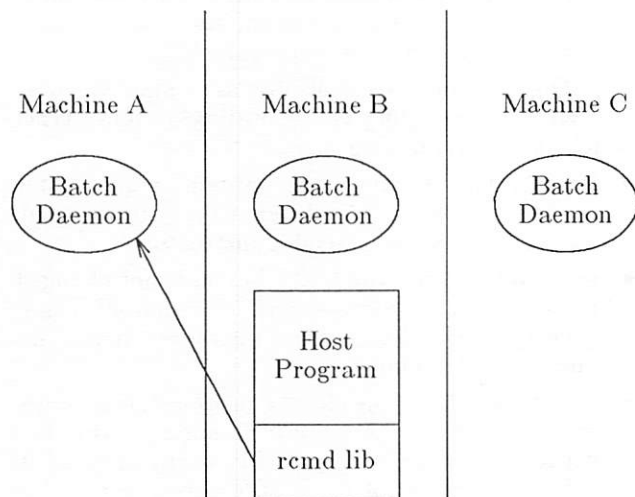


Figure 1. Host program makes remote execution request

broadcasts an accurate picture of the entire network. Each daemon then knows, concerning itself, whether jobs are waiting and whether the current job is active

or suspended; concerning other daemons, whether they are available for batching or not. Figure 1 shows a network of three machines, each with its own batch daemon. Machines A and C have no users, so they are available for batching. The running host program on machine B sends a remote execution request to the batch daemon on machine A. Thus is established a communication link between the host program on B and the batch daemon on A.

In Figure 2, the batch daemon has begun to respond to the remote execution request by forking a C-Shell which will execute the satellite. The host will then be able to communicate with its satellite, for the link between host and daemon is inherited by the shell

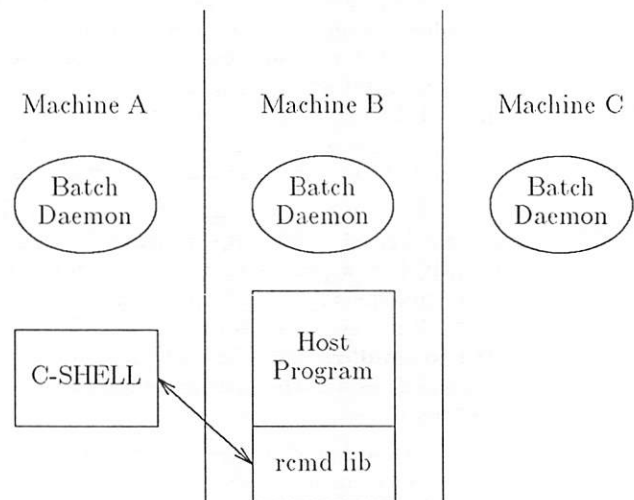


Figure 2. Machine A's batch daemon responds by forking a C-Shell

and then by any satellite the shell executes. Finally, the daemon closes its end of the communication link and awaits the next request.

The Addition of Process Cloning

In Figure 3 the daemon realizes a user has just logged on to machine A causing it to be in use; consequently, the satellite must be stopped. Under the old batching system, the daemon would have sent the satellite a nonignorable signal (SIGSTOP), and, despite the availability of machine C, left the satellite stopped on A. Figure 3 summarizes, therefore, the foremost shortcoming of the old system. Figure 4 shows how the addition of process cloning has overcome this disadvantage. There, having consulted its data base of status information and found that machine C is not in use, the daemon initiates cloning and migration by sending an entirely different signal, SIGTSTP, which can be caught. This signal triggers a handler in the rcmdio library which both saves the satellite's I/O state as a file and puts it to sleep. The batch daemon then saves to a second file the executable image, and to a third file the context of the sleeping satellite. These three files: I/O state, executable image, and context can be used to duplicate the

satellite on machine C. Once resumed, the original

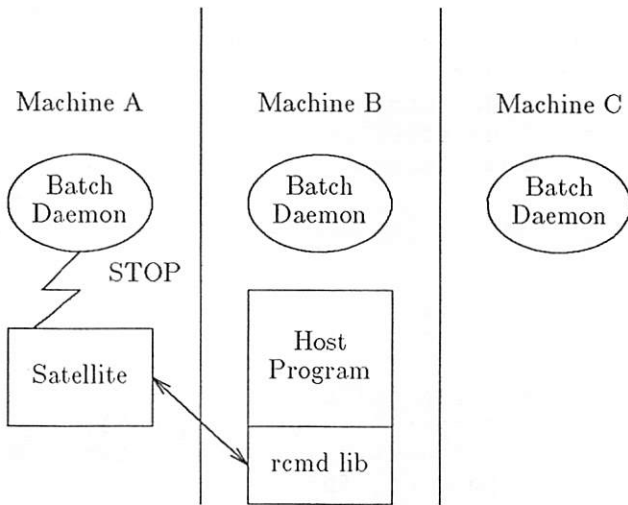


Figure 3. Old batching system: satellite is stopped when machine A becomes in use process will be destroyed.

Writing Clonable Programs

It's simple to ensure the clonability of a program intended to be executed as a satellite. The programmer need only refrain from certain system and library

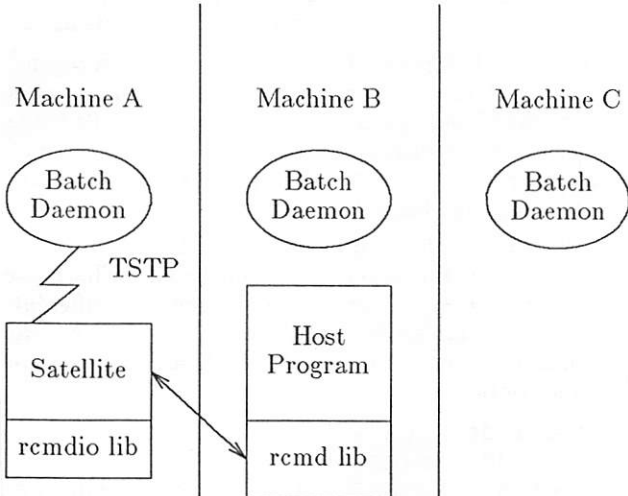


Figure 4. New batching system: satellite's handler is activated to begin cloning calls; include a macro file "migrate.h"; and link in the rcmdio library.

Excellent systems for process cloning systems have been written with different aims. Remote UNIX [2], a University of Wisconsin project, also links clonable processes with a special library to intercept certain UNIX system and library calls. Whereas our implementation executes these calls locally, Remote UNIX carries them out as "requests over the net to a shadow process on the originating machine." This process then executes the command and sends the

results back.

Each host in a distributed file system network may have a slightly different view of the file system. For example, in Sun Microsystem's NFS, each workstation has private root, /dev, and /tmp directories. Should a process migrate from a host on which it has been accessing files in a private directory, it will become confused when it cannot access those same files on its new host. Would be process migrators must decide whether to permit such problems to occur, disallow access to private directories, or guarantee continued access to them.

Remote UNIX, although designed for compute bound jobs (and thus little I/O), offers the distinct advantage of a uniform I/O environment. A process may migrate indefinitely among hosts in the network, yet its I/O calls are always carried out on the originating machine. In this way, a migrating process maintains the same view of the file system.

Unlike Remote UNIX, process cloning does not divide its jobs into two parts: one, which handles I/O and other calls, that runs on the originating machine; and the other, the compute bound user task that is to be migrated amongst possibly many hosts. Instead, satellites are migrated as complete, ordinary processes. While this approach does not provide the same uniformity, it eliminates satellite dependence on a shadow process for the execution of many commands; thus, I/O bound tasks can be accommodated as easily as those which are CPU intensive. In addition, the local execution of I/O calls guarantees that in use machines will not be impacted. Process cloning attempts to detect accesses to private directories and prevent processes which have violated this restriction from migrating further. This ensures a uniform I/O environment though it does not guarantee which host environment will be effective.

The macro file "migrate.h" overwrites certain calls, allowing them to be intercepted. This enables process cloning to record the I/O state of a process for later regeneration and also to detect violations of cloning restrictions such as private directory accesses. The macro file must be included after all UNIX include files and before any user include files, in order to prevent interference with the standard include files and make certain that none of the user's code eludes the necessary interception.

As already mentioned, process cloning does not support access to private directories such as /tmp or /dev under NFS. Also not supported are the UNIX system and library calls fcntl(), setbuf() et al, and pipe(); file locking; signals; multi-process jobs; inter-process communication (except that between the satellite and its host program); and indefinitely large stack sizes. Lastly, a program that makes use of its own process ID or the system clock will become confused, since these values will have changed unpredictably after cloning. In practice, users have found this seemingly large set of restrictions tolerable, as there are so many useful programs that can be written

which do not require these capabilities.

The process cloning implementation is amenable to expansion and provides facility to relax many of these limitations. Not only does it attempt to detect violations of restrictions, but it also can provide the user with a "crime report" which lists them if desired.

How programs are cloned

Process cloning views a program from three perspectives: its image, context, and I/O state. These three components are extracted from a running process that has been stopped, and are saved as three files; thereafter, the files can be reassembled into a duplicate of the original process. The image file contains the text (machine instructions) and preallocated data (such as variables). It alone is not enough to regenerate the process, since it does not know where the program was last running, nor the contents of the stack and registers. This is defined by the context file. Lastly, the I/O state file records which files the process had opened, how it had opened them, and where in each file the process had been looking. The I/O state also assists in reestablishing communication with the host program. Of these three components distilled from a process, I/O state is the first to be reactivated on a new machine. And although it is the most complex component to implement, it is the easiest to explore in detail. We will consider its maintenance, capture, and restoration.

The I/O State

Perhaps it is natural to think of the I/O state extraction as a procedure that is external to the process to be cloned. Unfortunately, any outside agent may have difficulty reaching into a process and understanding what it there discovers; it may also find itself following pointers into kernel structures. Still worse, depending on the technique employed, part of the operating system's state may have to be directly restored on the next host the process migrates to. This requires specialized knowledge of a kernel which will eventually change with operating system upgrades. A simpler, more portable approach is to have the process volunteer its own I/O state. Similarly, the process can itself later use this information to perform the I/O state restoration.

Maintaining the I/O State

It would be wonderful if users registered every command that affects the I/O state. This can be accomplished surreptitiously by replacing every I/O call with a version that first calls the desired routine and then performs the registration. However, to further reduce overhead, calls which read, write, or otherwise affect file offsets are not recorded. Instead, offsets are ascertained when the I/O state is captured.

Call substitution is accomplished with the "migrate.h" include file. Calls such as open(), close(), dup(), and chdir() are replaced with their counterparts in the rcmdio library. In this way, the crucial I/O calls are intercepted and their effects are

registered. If the user wishes, he will be mailed a report of the current I/O state on demand. In the following program, two files are opened, one to copy to, the other to copy from:

```
#include <stdio.h>
#include <fcntl.h>
#include "migrate.h"

main()
{
    FILE *fp;
    int    fd;
    char  ch;

    fp = fopen("foo", "w");
    fd = open("blort", O_RDONLY);
    while(read(fd, &ch, sizeof(char)) > 0)
        putc(ch, fp);
    io_state();
}
```

The names of the opened files and the way they are opened are registered by the substitute fopen() and open() calls in the rcmdio library. The io_state() function provides access to this information:

Current I/O state

```
fopen(): path "foo",    type "w",
          fd 3, dead 0
open():  path "blort", flags 0,
          mode 0, fd 4, dead 0
```

Here io_state() reports that file "foo" has been opened for writing (type "w") using the fopen() call and that "blort" has been opened for reading (mode 0) using open(). Also recorded are the file descriptors which resulted from these calls. The fields marked "dead" will be explained later.

In this simple example there can be no confusion as to where a file descriptor came from. There are only two of them, 3 and 4, and they are different. This lack of ambiguity is not always the case; yet the internal representation of the I/O state must not become confused:

```
#include <fcntl.h>
#include "migrate.h"
#define MESSAGE "Let's be confusing"

main()
{
    int    fd1, fd2, fd3, fd4, fd5;

    fd1 = open("foo", O_WRONLY|O_CREAT,
               0644);
    fd2 = dup(fd1);
    write(fd1, MESSAGE, strlen(MESSAGE));
    io_state();

    close(fd1);
    fd3 = open("snark", O_WRONLY|O_CREAT,
               0644);
```

```

fd4 = dup(fd3);
fd5 = dup(fd2);
io_state();

close(fd2);
close(fd5);
io_state();
}

```

The first I/O state report shows file "foo" opened for writing by open() and a dup()'d file descriptor:

```

Current I/O state
open(): path "foo", flags 513,
        mode 420, fd 3, dead 0
dup(): orig 3, copy 4

```

The io_state() call reports file "foo" with file descriptor 3 and its duplicate with descriptor 4. The "orig" field for the dup() entry lists the original descriptor from which the new descriptor, "copy" was made. The next action taken by the program is to close descriptor 3. If descriptor 3's open() entry now were to be removed from the internal representation, only descriptor 4's dup() entry would remain, known still as a duplicate of descriptor 3: what descriptor 3 itself was, that would be a mystery! To solve this problem, an extra field has been added to certain entries, the "dead" field, a binary flag which is set to one to indicate a dead but not forgotten entry. Dead entries are "buried" when no other entries depend on them. This technique obviates the confusion inherent in the following actions:

```

open(): path "foo", flags 513,
        mode 420, fd 3, dead 1
dup(): orig 3, copy 4
dup(): orig 3, copy 6
open(): path "snark", flags 513,
        mode 420, fd 3, dead 0
dup(): orig 3, copy 5

```

The closed descriptor 3 is marked dead. The new file "snark" is opened and assigned the first available descriptor, not coincidentally descriptor 3. Then the following dup() was executed:

```
fd4 = dup(fd3);
```

Despite that variable fd3 is 3, the internal representation correctly references file "snark," not "foo" nor its duplicate. The dup() made a copy of snark's descriptor with that first available, 5. This dup() entry follows snark's open() entry to indicate its dependence. Similarly, foo's dup() entry immediately follows its deceased open() entry. The following was the last dup() executed.

```
fd5 = dup(fd2);
```

The value 4 in the variable fd2 refers to foo's duplicate. Since a duplicate of foo is the same as foo, the descriptor resulting from this dup() is made to depend directly on foo. In other words, by the transitive law of file descriptors, the orig field for this dup() must be set to 3. Lastly, foo's duplicates are closed:

```

open(): path "snark", flags 513,
        mode 420, fd 3, dead 0
dup(): orig 3, copy 5

```

With no entries depending on it, foo's dead open() entry is finally laid to rest. All this magic is simply the result of representing the I/O state as a dependency graph. The io_state() function reveals some of the dependencies by placing entries after those which they depend on. This approach enables process cloning to handle easily much more complicated situations.

Capturing the I/O State

When a process is to be cloned, its I/O state is easily captured. The rcmdio library linked with the user's program contains a signal handler which responds to the signal SIGTSTP by dumping the program's I/O state to a file. If any violations of the cloning restrictions have occurred, a list of them is written to the I/O state file instead. A final aspect of process I/O state has not yet been mentioned: communication between this satellite process and its host program. This communication is conducted over a socket which, at any time, may have characters buffered up on either end. When cloning is about to occur, the program at each end of the socket reads and stores its buffered characters. In the case of the satellite, buffered characters are read and saved with the other information in the I/O state file.

Capturing the Image

The image of a process, unlike its I/O state, is captured externally. A program uses the ptrace() system call to peek into the process and copy out its text and data pages. Before this can occur, the process must tell UNIX it's ready to be traced. This occurs in the SIGTSTP handler. There, if no violations are discovered, the process readies itself to be ptrace()'d and then puts itself to sleep until which time the external program waits. As soon as the process sleeps, the external program inspects the I/O state file to be certain everything has proceeded normally. If it has, it attaches to the comatose process and constructs an image using ptrace().

Of course, since our application is process migration, the external program is the batch daemon. However, any external program having permission could trace the clonable program. For example, if process cloning were to be used to provide fault tolerant execution, the external program would then be a checkpoint.

The image is almost identical to the original executable file, but additionally includes data allocated at run time. Again, the image alone is insufficient for regeneration, because it does not indicate where execution left off or what the registers contained. Even were this not the case, without restoring the stack contents the history of stack frames would be lost. Among other disasters, this would cause variables declared on the stack to be undefined and, in general, chaos upon returning from subroutines. The program counter, stack pointer, all other registers, and

stack contents are contained in the context.

Capturing the Context

The context is captured by the same external program that preserves the image. Once the image has been extracted into an executable file, the context is similarly distilled into its file. The context file, however, is effectively a data file, for it will later be read by the regenerated process to restore its stack, registers, and program counter.

The Big Picture

Having investigated the three components of a process and how they are captured, we can now see the entire procedure. Figure 5 shows the progression of steps.

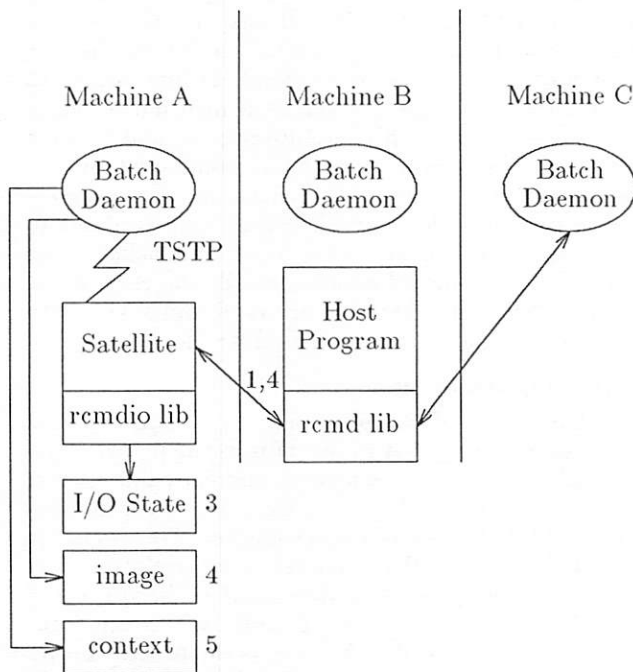


Figure 5. I/O state, image, and context capture

In this diagram, the batch daemon on machine A has already decided to migrate the satellite in its custody to the free workstation, machine C. The satellite has been sent a SIGTSTP signal, thus activating the SIGTSTP signal handler in its rcmdio library. The SIGTSTP handler reads the name of the next host, machine C, from a temporary file (not shown here) created by machine A's batch daemon. In step 1, the SIGTSTP handler informs the host program on B of the impending migration and passes along the name of the next host through the socket. Since this information is sent to the host program in the form of an out of bound message, a SIGURG handler in its rcmd library becomes active. The SIGURG handler then alerts machine C's batch daemon in step 2. To do this, the host program establishes a socket connection to machine C's batch daemon and provides the name of the migrating process to be expected. If this succeeds, the batch daemon on machine C then waits

for the I/O state, image, and context files to be deposited in the distributed file system. The socket connection will be inherited by the regenerated process. This step can be attempted repeatedly if machine C's batch daemon proves temporarily unreceptive. In the unlikely event of continued recalcitrant behavior, the process can even be migrated back to machine A as soon as it is captured. Meanwhile, the SIGTSTP handler in the satellite has been checking for violations of the cloning restrictions. If any were detected, appropriate error messages are written to the I/O state file and cloning is terminated after mailing a crime report to the user. In this case, the satellite would not migrate, but instead remain stopped until machine A was no longer in use. Assuming no violations occurred, the SIGTSTP handler generates an I/O state file in step 3. To complete the file, the communication state must also be saved. This is accomplished in step 4, when both the satellite and its host program read their buffered characters from the socket which joins them. After the host program has saved its data to a temporary file and the satellite its data to the I/O state file, the socket link between them is disconnected. Now that the SIGTSTP handler has fulfilled all of its obligations, it renders itself traceable and goes to sleep. Machine A's batch daemon, awaiting the satellite's slumber, now attaches to it for tracing. This daemon completes the capture by constructing the image and context files in steps 5 and 6 before killing the process. The batch daemon on machine C can now regenerate the process from the three files.

Resuming the Process

With the I/O state, image, and context files placed in the distributed file system, any machine with access to it will be able to restore the process. In figure 6, the free host, machine C, will be able to

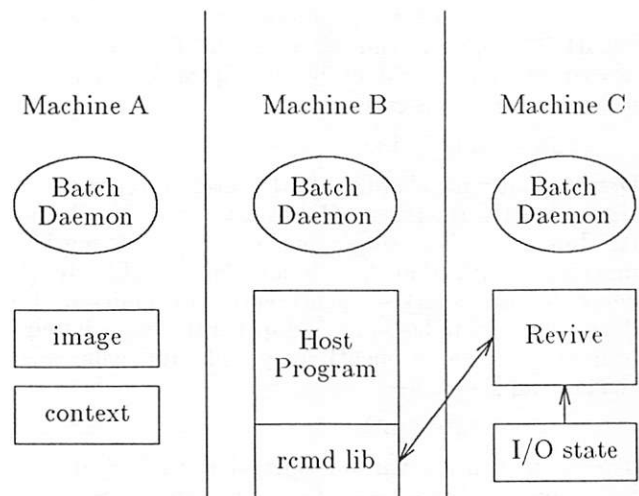


Figure 6. Regenerating the process on a free host

duplicate the process from the three files. The diagram shows the socket connection which was originally established to alert machine C of the

migration. This communication link is inherited by a program called "revive," which was started by machine C's batch daemon. Revive's purpose is to restore the I/O state before exec'ing the image over itself. In this way, the image inherits both the restored communication link and the I/O state. Revive reads the information in the I/O state file as a series of commands which direct it to reopen files, restore file offsets, et cetera. The last command in the I/O state file causes revive to restore communication with the host program on machine B. To accomplish this, the revive and host programs exchange the buffered data they saved earlier. Now each program has the characters that were buffered but unread by the other program at the time of cloning. The two programs exchange this data a second time so that the buffered characters are restored. With its work now complete, the revive program exec's the image over itself.

As was stated earlier, the image is an executable file whose contents are similar to the original executable which first started the process. When the image is loaded, it even begins execution at the same location as did the original executable. One of the covert effects of the migrate.h macro file not yet mentioned is to intercept the user's main() function by renaming it __main(). When the original executable is first run, the main() supplied in the rcmdio library discovers that it is, in fact, being run for the first time from checking a flag which is unset. This flag is then set before passing control and arguments to __main(). Later, after cloning, the image is executed and again finds itself in main(), but, discovering the set flag, realizes that it is an image and not the original executable, so it instead passes control to another rcmdio library routine, restart(). This function loads the stack, register, and program counter information from the context file into memory. The last library routine to be called is _rstart(), the only machine dependent code required by process cloning. This function is an assembly routine which restores the stack and register contents. Cloning is complete when _rstart() restores the program counter and execution continues where it left off.

Conclusion

The philosophy of process cloning shuns kernel modification, minimizes machine dependent code, shields the user from detail, and wherever possible coerces a process to participate in its own cloning. The result is a portable, durable, extensible, and, above all, simple implementation. When applied to a distributed batching system, process cloning provides significantly greater utilization of the unused CPU resource in a network.

Extensions under consideration include eliminating the stack size limitation and adding support for interprocess communication. Concerning the latter, our current implementation can represent socket related dependencies. Also under consideration are

future applications in fault tolerance. Conventional, sequential programs which run for days or indefinitely could be checkpointed by taking frequent snapshots. Should the host on which the program is running go down, valuable execution time could be saved by restoring the program from its last snapshot rather than restarting it from the beginning and losing days or weeks of CPU time. Similarly, parallel, distributed programs could be checkpointed at each node. Should a particular node go down (which could affect many dependent processes on other nodes), the process that it was executing could later be restored from its most recent snapshot and all dependent processes resynchronized to their respective snapshots. In this way, the cumulative execution time of all processes would be safeguarded.

Acknowledgements

I would like to thank Chris Gantz who shared many late nights making the implementation a reality and Sid Stuart, our group leader, who first proposed the project. I am also grateful to John McCoubrey for his guidance in writing this paper and everyone else who critiqued it, including Robert Silverman. Especial thanks to Dr. Thomas W. Doepfner of Brown University for his expert advice.

References

- [1] Robert D. Silverman, *Parallel Implementation of the Quadratic Sieve*, J. Supercomputing (to appear).
- [2] University of Wisconsin, "Remote UNIX- Turning Idle Workstations into Cycle Servers", USENIX Conference Proceedings Phoenix, Summer, 1987.

NOTES

NOTES

